View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Waterloo's Institutional Repository

Adaptive CPU Allocation for Resource Isolation and Work Conservation

by

Cong Guo

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2020

© Cong Guo 2020

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: Paolo Valente
Assistant Professor,
Dipartimento di Scienze Fisiche, Informatiche e Matematiche,
University of University of Modena and Reggio Emilia

Supervisor(s): Martin Karsten
Associate Professor,
Cheriton School of Computer Science, University of Waterloo

Internal Member: Ken Salem
Professor,
Cheriton School of Computer Science, University of Waterloo

Bernard Wong
Associate Professor,
Cheriton School of Computer Science, University of Waterloo

Internal-External Member: Rodolfo Pellizzoni
Associate Professor,
Dept. of Electrical and Computer Engineering, University of Waterloo

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Consolidating multiple workloads on the same physical machine is an effective measure for utilizing resources efficiently and reducing costs. The main objective is to execute multiple demanding workloads using no more than necessary resources while simultaneously maximizing performance. Conventional work-conserving resource managers are designed for this purpose. However, without adequate control, the performance of consolidated workloads may degrade dramatically or become unpredictable because of contention for shared resources. Hence, resource isolation should be enforced according to a sharing policy when there is resource contention among workloads, i.e., each workload should obtain a theoretical share of resources. In reality, it is challenging for state-of-the-art resource managers to achieve both resource isolation and work conservation simultaneously due to complex and dynamic workloads.

This thesis proposes adaptive resource allocation to address this sharing problem and studies CPU management as an example. A novel feedback-based resource manager is designed to perform adaptive allocation of CPU resources, taking into account each workload's requirements. First, an application-agnostic metric is proposed as the feedback signal, which can be used to measure the performance change of various applications in a non-invasive and timely way. Second, two alternative feedback-based algorithms are designed to search for the optimal resource allocation for each workload. The adaptive allocation is modelled as a dynamic optimization problem. The algorithms solve this problem by assessing performance changes in response to a change in resource allocation. The algorithms are demonstrated to be capable of handling complex and dynamic workloads. The resource manager proposed in this thesis uses these algorithms to determine the CPU allocation for multiple tenants. A prototype is implemented with four different sharing policies. For three common policies, the experimental evaluation confirms that the resource manager can achieve resource isolation and work conservation simultaneously, while the existing best-practice mechanisms cannot. Moreover, the resource manager can support a novel efficiency policy, which determines CPU sharing based on the overall system efficiency. In addition, a preliminary study shows that the feedback-based methodology for CPU management can be extended to control I/O bandwidth.

Acknowledgements

I would like to thank all the people who made this thesis possible. Foremost, I thank my supervisor, Professor Martin Karsten, for his guidance and support throughout my study. His supervision has helped me overcome the difficulties encountered during my research.

I also want to thank my committee members: Professors Ken Salem, Bernard Wong, Rodolfo Pellizzoni and Paolo Valente, for their patient review and valuable comments on my work.

I appreciate the help from my writing instructor Jane Russwurm and all my colleagues.

Finally, I am grateful for the love and encouragement from my parents.

Dedication

This is dedicated to my family.

Table of Contents

List of Figures x

List of Tables xiii

1 Introduction 1

1.1 Motivation 1

1.2 Contributions 3

1.3 Thesis Organization 5

2 Related Work and Background 6

2.1 Related Work 6

2.1.1 CPU Resource Management 7

2.1.2 Adaptive Resource Allocation 12

2.2 Background 19

2.2.1 Partitioning and Time-based Allocation 19

2.2.2 Resource Isolation 27

3 Metrics for Online Performance Measurement 33

3.1 Existing Performance Metrics 34

3.1.1 Application-Level Metrics 34

3.1.2 CPU Usage 35

3.2 User-level Instructions Per Time . . . 36
3.3 Relative Change . . . 38
3.4 Experimental Evaluation . . . 39
3.4.1 Evaluation of CPU Usage . . . 40
3.4.2 Evaluation of UIPT . . . 44

4 Adaptive CPU Allocation **50**
4.1 Problem Definition . . . 50
4.1.1 Performance Curve of A Stable Workload . . . 50
4.1.2 Problem Model for Dynamics . . . 52
4.1.3 Regulation Problem . . . 53
4.2 Hill Climbing . . . 54
4.3 Fuzzy Control . . . 59
4.4 Evaluation of Allocation Algorithms . . . 64
4.4.1 Stable Workload . . . 65
4.4.2 Unsaturated Workload . . . 68
4.4.3 High-Frequency Dynamic Workload . . . 71
4.4.4 Low-Frequency and Large-Magnitude Dynamic Workload . . . 75

5 CPU Resource Manager for Multiple Tenants **77**
5.1 System Overview . . . 78
5.2 Sharing Policies . . . 80
5.2.1 Proportional CPU Sharing . . . 80
5.2.2 Priority-Based CPU Sharing . . . 82
5.2.3 Resource Guarantee for Multiple Tenants . . . 82
5.2.4 Efficiency-Based CPU Sharing . . . 83
5.3 Evaluation of Resource Manager . . . 86
5.3.1 Experiment Design . . . 86

5.3.2 Proportional Sharing Policy 88
5.3.3 Priority Policy 93
5.3.4 Guarantee Policy 101
5.3.5 Efficiency Policy 104

6 I/O Bandwidth Management **110**
6.1 I/O Bandwidth Control 111
6.2 Experimental Evaluation 113
6.2.1 UIPT 113
6.2.2 Adaptive I/O Bandwidth Throttling 114

7 Conclusion **118**
7.1 Summary 118
7.2 Future Work 119

References **121**

APPENDICES **129**

A Evaluation of the Fuzzy Control Algorithm **130**
A.1 Experiment Results 130
A.1.1 Stable Workload 130
A.1.2 Unsaturated Workload 133
A.1.3 High-Frequency Dynamic Workload 135

B Evaluation of the Resource Manager **137**
B.1 Proportional Sharing with Different Workloads 137

List of Figures

2.1 Performance Difference of TPC-E Workload with Partitioning and Quota . 21
2.2 Cycle Usage with TPC-E Workload 22
2.3 Atomic Instruction Overhead with TPC-E Workload 23
2.4 Memcached Performance with Quota and Partitioning 24
2.5 CPU Utilization with Write-heavy Memcached Workload 25
2.6 Estimated Cycles in a Linux Spin Lock Function with Write-heavy Memcached Workload 26
2.7 DBMS Performance with Weight vs. Partitioning 27
2.8 CPU Usage of the CPUhog Workload with the Lower Weight 29
2.9 CPU Usage of the TPC-E Workload with the Lower Weight 30
2.10 Results with Different Workloads 31

3.1 CPU Usage for Database Workloads 42
3.2 CPU Utilization for Database Workloads 43
3.3 UIPT for TPC-E Workload 45
3.4 UIPT for TPC-H Workload 45
3.5 UIPT for Memcached Workload 46
3.6 UIPT for TPC-E Workload in a VM 48
3.7 UIPT for TPC-E Workload in a Docker container 49
3.8 UIPT for TPC-E Workload with Different Quota Values 49

4.1 Architecture of the Allocation System 55

4.2 Architecture of the Controller 60
4.3 Membership Functions 62
4.4 Core Allocation with Hill Climbing for Stable TPC-E Workload 66
4.5 Results with Hill Climbing for Stable TPC-E Workload 67
4.6 Results with Hill Climbing for Canneal Benchmark 70
4.7 Core Allocation with Single-Timescale Hill Climbing for Stable TPC-E Workload 72
4.8 Core Allocation with Hill Climbing for High-Frequency Dynamic TPC-E Workload 73
4.9 Results with Hill Climbing for High-Frequency Dynamic TPC-E Workload 74
4.10 Results for Low-Frequency Dynamic TPC-E Workload 76

5.1 Resource Manager Architecture 78
5.2 CPU Usage and Allocation between Stable Workloads with Weights of 3:1 91
5.3 Throughput of MariaDB/TPC-E Workload 92
5.4 CPU Allocation between Dynamic Workloads with Weights of 3:1 94
5.5 Throughput of MariaDB/TPC-E Workload 95
5.6 Results for Stable Workloads with Different Priorities 97
5.7 Core Allocation with Dynamic Workloads 100
5.8 Primary MariaDB/TPC-E Throughput 100
5.9 Core Allocation between Stable Workloads with Guaranteed CPU Resources 103
5.10 Throughput of Memcached/Mutilate Workload 103
5.11 Core Allocation between Dynamic Workloads with Guaranteed CPU Resources 105
5.12 Throughput of Memcached/Mutilate Workload 105
5.13 Efficiency with Stable Workloads 107
5.14 Efficiency with Dynamic Workloads 109

6.1 UIPT for Etcd Put Workload 114
6.2 Etcd Workload: Start from 10 Mbps 115

6.3 Etcd Workload: Start from 60 Mbps 116

A.1 Core Allocation with Fuzzy Control for Stable TPC-E Workload 131

A.2 Results with Fuzzy Control for Stable TPC-E Workload 132

A.3 Results with Fuzzy Control for Canneal Benchmark 134

A.4 Core Allocation with Fuzzy Control for High-Frequency Dynamic TPC-E Workload 135

A.5 Results with Fuzzy Control for High-Frequency Dynamic TPC-E Workload 136

B.1 CPU Usage of the CPUhog Workload with the Lower Weight 138

B.2 CPU Usage of the TPC-E Workload with the Lower Weight 139

B.3 Results with Different Workloads 140

List of Tables

2.1 FC2Q Control Rules 17

3.1 Parsec Correlation Results 47

4.1 Fuzzy Rules 62
4.2 Algorithm Parameters 65

5.1 Current and Baseline Mechanisms for Different Policies 87
5.2 Swaptions Execution Time with A Stable TPC-E Workload 92
5.3 Swaptions Completion Time with A Dynamic TPC-E Workload 95
5.4 Secondary MariaDB/TPC-H Completion Time 100
5.5 Swaptions Completion Time with Stable Memcached/Mutilate Workload . 103
5.6 Swaptions Completion Time with Dynamic Memcached/Mutilate Workload 105

Chapter 1

Introduction

1.1 Motivation

The operators of computing facilities, such as cloud service vendors, universities and high performance computing centres need to continuously improve service as well as reduce costs. Especially cloud service vendors are driven by a highly competitive business. Workload consolidation, which combines multiple workloads onto fewer physical servers is an essential path to utilizing more of the existing resources and to reducing the costs of infrastructure. A major aim of workload consolidation is to keep resources busy by combining as many workloads as possible onto a physical server. To better understand the importance of work consolidation, it is helpful to recall how operating systems on a single machine schedule tasks. CPU schedulers in operating systems consolidate threads on the machine's CPU via time-multiplexing to keep the CPU busy. If a CPU scheduler ensures that processors are not idle when there are threads ready for execution, it is referred to as a work-conserving scheduler. Work-conserving scheduling can be generalized to work-conserving CPU sharing, i.e., not leaving idle CPU resources when the requirements of certain workloads are not yet met.

Currently, single machines are arranged together into cloud infrastructure to serve a large number of heterogeneous and dynamic workloads. Consolidating workloads and effectively utilizing existing machines are still indispensable tasks. Despite the large quantity of resources in the cloud, many of these resources are not utilized all the time. A significant reason is that computing resources may be reserved for particular workloads but these resources are not always being utilized. Several previous studies report that the average server utilization in most data centres is less than 50%, and for Amazon Web Services,

the utilization is even lower than 20% [5, 7, 17, 48, 63, 69]. Hardware infrastructure is the largest fraction of total cost of ownership (TCO) for cloud service vendors. A recent report reveals that increasing utilization by a few percentage points in a large data centre can lower the TCO significantly [5].

Workload consolidation is not simply putting workloads onto the same machine. If too many workloads are consolidated together, even though server utilization is improved, application performance may degrade dramatically or become unpredictable because of contention for shared resources. Resource managers control how resources are shared among workloads by defining sharing policies. Each workload has a *theoretical share* based on the sharing policy. For instance, with the proportional sharing policy, a workload's theoretical share is a fraction of the total CPU resources based on relative weights; with the strict priority policy, the theoretical share of a high-priority workload is the same as its requirement. Here the CPU requirement of a workload refers to the CPU resources required by its software system to achieve the optimal performance. An effective resource isolation must meet two conditions: (i) when a workload's CPU requirement is less than or equal to its theoretical share determined by the sharing policy, the workload's requirement must be met; (ii) when a workload's requirement is more than its theoretical share, the workload should be able to consume at least its theoretical share of CPU resources. Thereby, when multiple workloads require CPU resources in excess of their theoretical shares, and CPU resources are not sufficient to meet all the requirements, resource managers should ensure that no workload is degraded because its CPU share is occupied by other workloads. On the other hand, if a resource manager stays work-conserving while enforcing resource isolation, a workload can consume more CPU resources than its theoretical share when there are otherwise idle CPU resources.

However, it is challenging for current work-conserving resource managers to enforce resource isolation. The actual CPU resources each tenant workload can consume depend on the presence and execution patterns of other co-located workloads [10, 67]. Some sharing mechanisms limit the maximum CPU resources a workload can utilize. Via non-overlapping limits, a resource manager can protect workloads from being affected by each other. Unfortunately, a static limitation mechanism prevents workloads from utilizing idle resources. In other words, a static limitation mechanism is not work-conserving.

An ideal solution to workload consolidation should be able to effectively restrict CPU usage when contention occurs, and dynamically adjust the limits according to the actual requirements of workloads. When resources are insufficient to meet all requirements, resource isolation should be enforced using the limits calculated based on the sharing policy. Meanwhile, when some workloads' requirements decrease, an ideal solution ought to allow free resources to be utilized by other workloads that still require more resources.

It is, nevertheless, difficult to determine the resource requirements of workloads given their complexity and variety. Current resource managers rely on users such as job submitters and application operators to configure their resource requirements. The users' configurations specify the type and quantity of resources needed. Once a configuration is established, it rarely changes. In practice, users may not be able to accurately specify resource requirements of their workloads, because real-world workloads are highly complex and different [63]. Moreover, these real-world workloads are usually dynamic, and thus a fixed resource configuration is not suitable all the time. Therefore, users are likely to over-provision their workloads, which can lead to a waste of resources and a low utilization.

In this thesis, CPU resources are allocated to workloads based on real needs of the workloads instead of user configurations. The proposed resource manager does not reserve more CPU resources for workloads than their requirements. Idle CPU resources can be utilized by unsatisfied workloads beyond their theoretical shares and thus work conservation is achieved. Another goal of this work is to enforce resource isolation according to sharing policies. If the requirement of each workload is known, an arbitrator can determine the allocations based on a specific policy. Thereby no workload occupies resources that should be used by others.

1.2 Contributions

CPU-management mechanisms in the widely used Linux system are investigated in this thesis through an experimental study. The results show that these mechanisms cannot simultaneously enforce resource isolation and support work conservation for consolidated workloads. This thesis presents a feedback-based resource manager to address this sharing problem. The resource manager automatically detects the CPU requirements of tenant workloads, thereby dynamically adjusting CPU sharing based on the requirements and sharing policies. The following specific contributions are made:

- *An application-agnostic fine-grained metric is proposed for online feedback measurement.*

A feedback-based solution needs to measure changes in application performance consequent to changes in CPU allocation. This thesis proposes an application-agnostic fine-grained metric named user-level instructions per time (UIPT). UIPT is calculated based on hardware-level measurements, and not specific to any application. It has a good correlation with various application-level performance metrics. The change in UIPT can be used to estimate the change in these application-level metrics. The measurement of UIPT does not require any software integration of applications or modification in the OS kernel.

In addition, with UIPT, meaningful performance feedback can be measured even within sub-second intervals, meaning that the adaptive allocation algorithm can adjust CPU allocations in a timely way.

• *Two feedback-based algorithms are presented to dynamically allocate CPU resources to a workload adaptive to the workload's changeable requirement.*

The algorithms are based on a simple generic performance model. For a stable workload with a stable request rate and static characteristics, as CPU allocation increases, the application performance first increases but levels off or even decreases after a certain point. Moreover, since real-world workloads are often time-varying, the performance curve may move over time. Therefore, the allocation problem is modelled as a dynamic optimization problem. For a stable workload, the algorithms need to search for the best allocation, i.e., the workload's requirement and overcome problematic regions in the performance curve, e.g., a plateau area. It is also necessary for the algorithms to address dynamics with different magnitudes and frequencies.

Two algorithms are proposed to solve this dynamic optimization problem. The first algorithm measures average performance changes at two time-scales, and applies a simple hill-climbing algorithm at each time-scale. The second algorithm uses a fuzzy control algorithm to search for the best CPU allocation. The experiments with different applications show that both algorithms can find proper allocations for stable workloads and adapt to dynamic workload changes.

• *A resource manager that supports multiple sharing policies is designed based on adaptive CPU allocation.*

Based on the UIPT metric and the allocation algorithms, this thesis presents the design and software prototype of a feedback-based resource manager. This resource manager dynamically allocates CPU resources based on the workloads' requirements and enforces resource isolation according to sharing policies. Three sharing policies in reality - proportional sharing, priority, and guarantee - are supported. In all three scenarios, the resource manager ensures that no tenant workload uses excess CPU resources when competition happens. Moreover, a novel policy supported only by this resource manager is introduced. This policy allocates CPU resources to the most efficient application regardless of the application type and workload dynamics. The prototype runs entirely outside the kernel and needs neither kernel nor application modifications.

In addition, the methodology for adaptive CPU allocation is extended to control the I/O bandwidth limit. UIPT is independent of resource types, and can be used to measure the performance of I/O-bound workloads. This thesis applies the UIPT-based allocation algorithms to I/O bandwidth control. The preliminary investigation shows that the

feedback-based methodology for CPU allocation can be extended to manage other types of resources if there is an effective static allocation interface like Linux cgroups.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 first surveys previous work relevant to the topic of this thesis, and then investigates existing CPU management mechanisms via an experimental study. The following three chapters show how a feedback-based resource manager for consolidation of various workloads is built. Chapter 3 introduces the application-agnostic metric UIPT to measure the application performance online, which provides the basis for generic adaptive CPU allocation. Chapter 4 proposes two adaptive CPU allocation algorithms based on UIPT and uses experiments to show their effectiveness. The basic allocation algorithms are extended to a resource manager that simultaneously achieves resource isolation and work conservation. Chapter 5 shows how the CPU manager performs with four different sharing policies. In Chapter 6, the same methodology is applied to I/O bandwidth control. Finally, Chapter 7 summaries this thesis and outlines potential future research.

Chapter 2

Related Work and Background

CPU management is an essential feature in resource managers[1], yet current resource managers struggle to properly support resource isolation and work conservation simultaneously. This thesis proposes making use of adaptive CPU allocation to address this problem. Although there have been various research efforts on adaptive resource allocation, most are either not designed for the problem studied in this thesis or support specific applications only. This chapter has two primary goals: 1. To survey previous work related to CPU management and adaptive resource allocation. 2. To investigate the limitations of existing CPU management systems experimentally.

2.1 Related Work

The thesis focus is on how to enforce resource isolation while making full use of available resources. This section first surveys how CPU resources are managed in production environments, especially how resource isolation is enforced. Next, previous studies on adaptive CPU allocation based on workload requirements are reviewed.

[1] In some proposals, the CPU management system is called a CPU scheduler. This thesis uses *resource manager* to refer to a CPU management system in general, but when specific work is discussed, the terminology provided in the work is used.

2.1.1 CPU Resource Management

This section provides a brief summary of the literature on CPU management to clarify the positioning of this thesis. The schedulers and resource managers discussed in this section are deployed in production environments to manage a single machine or a cluster of machines. Particular focus is given to how these systems share CPU resources among multiple workloads on each local machine.

CPU Scheduling

The Operating System (OS) is responsible for the resource management on a single computer system. In an operating system, a CPU scheduler allocates CPU resources to workloads in the temporal and spatial dimensions. On a uniprocessor machine, a CPU scheduler allocates the sole CPU to tasks via time-multiplexing. When multiple CPU cores are available, a CPU scheduler is also responsible for balancing load on these cores. A general-purpose CPU scheduler aims to let runnable tasks have fair access to the CPU resources. Meanwhile, it is also desirable for a CPU scheduler to be work-conserving, i.e., never leave the CPU idle if there are runnable tasks.

Practical CPU schedulers attempt to approximate an ideal fair scheduling discipline called Generalized Processor Sharing (GPS) [60]. GPS shares CPU or network resources in perfect proportion to tasks' or flows' weights. Nevertheless, GPS cannot be implemented in reality. Different categories of algorithms have been developed to approximate GPS, such as round robin and fair queueing [18, 37, 46]. These fair scheduling algorithms show two common characteristics. First, each task is given a CPU time slice, after which the task is scheduled again. Second, if a task does not use up its time slice because of, e.g., waiting for an I/O operation or being blocked by a synchronization function, unused CPU time will not be compensated later.

On a multi-core system, a CPU scheduler can run an instance of the GPS-based scheduling algorithm on each core separately to avoid scalability issues. Yet fairly allocating CPU on a per-core basis does not necessarily result in global fairness, and thus these schedulers must rely on load balancing to evenly distribute threads over multiple cores [57, 79]. The global fairness among threads is undermined if load is unbalanced. Threads on a core with higher load obtain less CPU than those on another core with less load even if they have the same weight.

A practical example of the approximation algorithm is the Completely Fair Scheduler (CFS) in the Linux system [57]. CFS employs a work-conserving scheduling algorithm

which sorts all runnable tasks by their virtual runtime in a red-black tree (called a runqueue). Here the virtual runtime is the weighted amount of time a task has spent on a processor. In CFS, the input task priority is converted to a weight to calculate virtual runtime. The leftmost task in the runqueue has the smallest virtual runtime. Each running task obtains a CPU time slice. Each time slice is a fraction of the configured scheduling period, and the fraction is determined by each task's weight over the total weight of all tasks in the same runqueue. When a running task's time slice expires, if another runnable task becomes the leftmost task, the scheduler picks up this new task and preempts the current task. If a task yields the CPU before its time slice expires, it can get an advanced position in the runqueue due to its relatively shorter virtual runtime. However, unused CPU time will not be compensated.

CFS prevents a task from having a virtual runtime much lower than the virtual runtime of other tasks waiting to be scheduled. If that were to happen, the task with the low virtual runtime could run for a long time and starve other tasks. Therefore, when a task is created, the task starts with a virtual runtime equal to the maximum virtual runtime of the tasks waiting in the runqueue. When a task wakes up after sleeping, its virtual runtime is updated to be at least equal to the minimum virtual runtime of all other waiting tasks. Nevertheless, the CFS design still has a potential fairness issue. CFS tends to favour programs with a higher number of tasks, because the allocation is based on distributing the CPU time equally among tasks (threads) in the system instead of among programs. This issue is demonstrated by another experimental study [74].

On multi-core platforms, CFS is more complex than the basic algorithm. It keeps a local runqueue for each core due to scalability concerns. In the presence of per-core runqueues, tasks in each runqueue must be kept balanced for the scheduling algorithm to work properly. Without load balancing, for example, if two high-priority tasks were occasionally on the same core while a single low-priority task were on another core, these high-priority tasks would get less CPU time than the low-priority task. Therefore, CFS periodically runs a load-balancing algorithm that keeps the queues roughly balanced. Additionally, on a multi-core platform, while tasks can access all the CPU cores by default, Linux can set CPU affinity of tasks to restrict the specific cores they can use. CFS checks the CPU affinity when it places a task into a runqueue.

Besides CPU schedulers, Linux provides an interface that controls allocation and isolation of resources among user-defined groups of tasks [9]. This Linux cgroups (control groups) system is widely used in resource managers. A cgroup in the Linux cgroups system is a set of tasks, and the system defines each cgroup's access to different resources through different subsystems. Currently Linux cgroups can manage CPU, main memory, disk I/O, network, and combinations of these resources. The cgroups system can be

used to control resources for natively executing processes, containers, and virtual machines (VMs). The resource configuration of the tasks in a control group can be modified through parameters in subsystems.

The Linux cgroups system has two subsystems, *cpuset* and *cpu*[2], to control CPU allocation. The cpuset subsystem assigns individual CPU cores and NUMA memory nodes to cgroups. The parameter *cpus* in the cpuset subsystem specifies the physical cores that can be used by tasks in a cgroup. The core allocation can be non-exclusive or exclusive.

The cpu subsystem provides two different mechanisms, *quota* and *weight* to manage the sharing of CPU time among groups of tasks. The quota mechanism defines the maximum amount of CPU time that a cgroup can consume during a reference period. The quota mechanism uses two parameters in the cpu subsystem, *cfs_quota_us* and *cfs_period_us*, to define the maximum amount of CPU time and the reference period. Once the tasks in a cgroup have used up all the time specified by the quota, they are not allowed to run during the remainder of the period. When the quota is larger than the period, the tasks can use multiple cores. For example, when cpu.cfs_quota_us is 200,000 and cpu.cfs_period_us is 100,000, tasks in this cgroup can utilize CPU time equivalent to two full cores during each period. The quota mechanism limits the CPU usage of a workload strictly even though there are idle CPU resources. Hence this mechanism is not work-conserving.

The weight mechanism allocates CPU time to each cgroup proportional to their relative weights. An integer parameter, *shares*, specifies the relative share of CPU time available to a cgroup. For example, tasks in a cgroup that has cpu.shares set to 2048 can receive twice as much CPU time as tasks in a cgroup whose cpu.shares is set to 1024. When the tasks are in a cgroup under both cpu and cpuset subsystems, the relative share of CPU time applies to cores which are specified in the cpuset. Otherwise, this parameter applies to all cores in the system. The weight mechanism determines the share of each workload. Yet this mechanism is work-conserving. A workload can use more CPU resources than its share when there are otherwise idle CPU resources.

Theoretically, weighted fair scheduling is designed to support resource isolation and work conservation simultaneously. However, real-world OS schedulers have non-trivial overhead in practice, especially when taking into account fairness and effectiveness across multiple cores. Discretization (time-slice-based scheduling) is used to limit the computational overhead. Linux CFS implements weighted fair scheduling in principle, but practical experiments (c.f. Section 2.2.2) show the issues of this scheduler with respect to proportional sharing, which are potentially caused by the cumulative effect of discretization errors.

[2]This thesis uses terminology from cgroups version 1, but the same mechanisms are still used in cgroups version 2.

Another popular OS scheduler, FreeBSD's ULE scheduler [65] does not implement weighted fair scheduling at all.

In contrast, an upper bound on resource usage is relatively easy to implement. The *quota* mechanism introduced above is the implementation of usage limit in the Linux kernel. FreeBSD has a similar mechanism for resource usage limit [64]. This study is based on the effective usage limit. This thesis does not attempt to improve the inherent capabilities of a multi-core scheduler, but explores techniques to attain desirable resource allocation properties while accepting the limitations of real-world schedulers.

Cluster Management

For large-scale distributed computing, resource managers are designed to manage a cluster that consists of a large number of compute nodes (physical machines). The workloads running on a cluster vary from long-running computations over days to many short tasks taking minutes or seconds. Technologies, such as batch jobs, virtual machines, and containers are used for these workloads to share physical resources. Generally speaking, resource managers have a component that schedules tasks based on configurations and selects specific compute nodes for the tasks. After a task is launched on a compute node, another component deployed on each node performs local scheduling and enforces resource isolation. There have been quite a few studies on the scheduling and node assignment at the cluster level. Batch job schedulers calculate an overall priority for each job based on different policies, and then execute jobs in the priority order to fulfill cluster-wide goals such as fairness and utilization [27, 35, 77]. Resource managers in VM or container environments first filter out ineligible compute nodes based on conditions in requests describing resources and policies, and then rank nodes that survived filtering by calculating node scores according to certain global goals [26, 54, 72]. Cluster-level resource management is beyond the scope of this thesis. The focus of this thesis is on the local resource management, particularly how to share CPU resources in a work-conserving manner while enforcing resource isolation.

Next this section introduces how the local resource management is done in cluster resource managers. To start with, a batch job scheduler harnesses massive computing infrastructure to process various types of jobs. The resources each job requires are specified by a user request. A job scheduler assigns each job to a node with sufficient available resources. Resource usage limits are defined to constrain the amount of different resources a given job can consume. These limits are generally proportional to the resources requested. Each job scheduler has a component to track the resource usage of jobs on each node. When a resource usage limit is violated, a pre-defined action is taken. The pre-defined actions

include suspending, cancelling or requeueing the job [35]. Some job schedulers kill the job immediately [27, 77].

Virtualization technologies are commonly used to host emulated servers on a large cluster of physical machines. Each VM specifies a bundle of resources requested by a user. OpenStack is a widely-used open source orchestration system for VMs [55]. Its compute scheduler Nova is responsible for managing CPU resources [54]. Theoretically, a virtual CPU core in a VM corresponds to a physical core, but CPU resources are usually overcommitted to achieve a higher utilization. CPU scheduling and resource isolation on each physical node depend on the local operating system or hypervisor. Technologies in an operating system like Linux CFS and cgroups are discussed above. A hypervisor (also named virtual machine monitor) is responsible for creating and running VMs. Hypervisors use technologies similar to those in operating systems. For example, the default credit scheduler in the open source Xen hypervisor allocates CPU time slices to virtual CPUs, assigns relative weights to different VMs, and sets up usage "caps" of VMs [10]. Another example of VM technology is libvirt, which is a toolkit supporting multiple hypervisors to manage VMs [29]. The libvirt API employs Linux cgroups or a similar limiting mechanism on other operating systems to isolate resources.

Containers are another type of widely used virtualization technology. The container is based on OS kernel features, and provides an OS-level isolated virtual environment without hardware emulation. Some container technologies such as the Docker platform [19] and Linux containers (LXC) [11] are designed to execute long-running applications. Based on Docker, the open source Kubernetes system [26] automates application deployment and scaling. Kubernetes uses Linux cgroups for resource isolation on each host. With Kubernetes, users can configure three quality of service (QoS) classes for applications. These QoS classes are actually three priority levels, which are defined by resource "requests" and "limits" in the Kubernetes configuration. These priorities are implemented via the weighted time-based allocation mechanism in the cpu subsystem of Linux cgroups.

Google presents a commercial cluster scheduler named Borg for container management in [72]. Borg is a logically centralized controller. Tasks are assigned different priorities, and tasks with higher priorities can preempt those with lower priorities. Borg has its own implementation of application containers [49]. An agent process runs on each machine to execute tasks in containers. This kind of container is based on Linux cgroups. The Borg paper reports two interesting facts. First, Linux CFS requires substantial tuning to support both latency-sensitive workloads and server consolidation. Second, the weight is insufficient to emulate multiple priority levels in practice. This paper mentions that Borg has a mechanism that dynamically adjusts the resource limits of latency-sensitive tasks to avoid the starvation of batch tasks, but no details are given about how this mechanism is

implemented.

Specialized resource managers are designed for containers in big data systems. Big data workloads need a large number of short computations taking seconds or minutes to process enormous quantities of data. Big data systems, which are distributed systems, run these short tasks in containers. Mesos [31] and YARN [70] are two popular open source resource managers of this kind. Mesos slaves on each node run tasks in containers, and monitor the execution and resource usage. Mesos slaves employ Linux cgroups or POSIX rlimits [64] as resource isolators to limit CPU usage of tasks. YARN has a similar component named NodeManager. When tasks are running in containers on nodes, the NodeManager employs the quota mechanism in Linux cgroups to limit CPU usage of each container. The quota value is calculated based on the ratio of the virtual cores of the container and the available physical cores on the machine.

To sum up, the design of cluster resource managers in practice has tended to focus on node selection rather than local resource isolation on each node. CPU resources are managed using weighted time-based allocation to support proportional sharing and priority. The underlying OS schedulers and time-based allocation mechanisms are usually work-conserving, but resource usage limits are necessary to enforce resource isolation [67]. The resource usage limits depend on resource requirements configured by users. However, users do not necessarily understand the resource requirements of complex applications and the variations of their workloads.

2.1.2 Adaptive Resource Allocation

This thesis studies CPU management among multiple tenants based on dynamically allocating CPU resources to tenants according to their requirements. In reality, the resource requirements of most tenant applications do change over time. Static usage constraints are not feasible all the time and may result in underutilized resources. There have been multiple previous attempts to dynamically adjust resource allocations to match the time-varying requirements of workloads. This section discusses common methods and typical papers on adaptive CPU allocation in the literature. Most of these studies aim to achieve the target performance of primary applications in the form of service level objectives (SLOs) or QoS guarantees, while leaving as many resources as possible to other secondary applications.

Prediction-based Approaches

Predictive modelling has been used in the context of adaptive resource allocation. For instance, some proposals train a performance model to predict the application performance with a given workload when resource allocation varies [43, 76]. With such a model, resource managers can determine resource allocation based on the target performance of applications. This kind of approach usually does not monitor application performance during runtime.

SmartSLA [76] employs adaptive resource allocation to minimize the total SLA (service level agreements) penalty costs. This work applies mature regression techniques to train a system model for predicting the performance of database systems with different resource allocations. The method requires preparatory experiments with different configurations and offline training. Different workloads may have different models even though the modelling process is the same. With the models, a decision module determines the necessary allocations and solves resource conflicts between tenants based on predicted SLA penalty costs.

Predictive system models can be constructed based on historical system logs and statistics. An automatic job provisioning system called AROMA has been developed for Hadoop in this way [43]. In the offline phase, AROMA analyzes Hadoop log files and resource utilization data and classifies past jobs into different groups. Jobs in the same group exhibit similar resource utilization patterns. Next AROMA applies the support vector machine (SVM) technique [68] to learn the performance model for each group of jobs. The model estimates the completion time of jobs with different input data sizes, resource allocations and configuration parameters. In the online phase, AROMA first measures the resource utilization signature of a newly submitted job by running it in a staging cluster of small VMs with default configuration parameters. AROMA matches the signature with known utilization patterns and finds the best fit model to use. Based on the model, AROMA calculates the number and type of VMs to be allocated, and chooses appropriate configuration parameters to meet the completion deadline of a job at the minimum cost.

These systems based on performance models perform well in their respective particular experiments. However, the models are specific to a particular application or workload, such as a query set or a Hadoop job. Moreover, the training process of these models tends to take a long time before a sufficiently accurate model is learned. Several attempts have been made to reduce the modelling cost, but the model accuracy is traded off.

A resource manager, Quasar [17], first profiles workloads on a few servers for a short period of time and then uses fast classification techniques to estimate the influence of

resource allocation and server assignment on performance. The result of classification is a prediction of application performance with different resource allocations, server types and the interference from other workloads. Based on such a performance model, Quasar makes allocation decisions to meet the performance constraints provided by users. This system still needs to adjust classifications to fit different workload types. Quasar not only uses the performance predictions, but also monitors actual performance to detect prediction errors and workload changes. When its monitoring detects performance deviations from the given constraints, Quasar does a reclassification and adjusts resource allocation and assignment.

Ernest is another study of low-overhead performance prediction [71]. The goal of this work is to determine the number and type of machines for large-scale analytical workloads based on performance predictions. Quasar profiles workloads using the first few tasks, while Ernest runs the entire job on small sample datasets and uses the training data to create a performance model. According to the paper, the prediction accuracy is not sufficient to enforce strict SLOs.

As an alternative to training performance models, time-series-based approaches use historical data to build black-box models to forecast resource requirements of applications. CloudScale [66] employs resource-usage time series to predict the resources required to enforce SLO conformance. In CloudScale, historical traces and online usage measurements are fed into the online prediction scheme developed in [25]. Signal processing techniques are used to extract the signature of current usage measurements. If the signature matches any signature of repeating patterns, the predication is performed based on that pattern. For workloads without repeating patterns, this system uses a discrete-time Markov chain with a finite number of states to predict the resource usage in the short term. CloudScale avoids under-estimation errors by adding a small extra value to the predicted resource requirement, and adjusts this padding value based on application-level SLO feedback. Thereby, this system requires each tenant's SLO as input and monitors application-level performance to detect SLO violations. When there are scaling conflicts between tenants, CloudScale uses virtual machine migration to solve them.

A time-series model can be adjusted recursively based on online measurements. However, to make accurate predictions for a short time scale, time-series-based approaches usually require that specific patterns are present in the historical data at that time scale, which may not be the case. The lack of such patterns impacts the accuracy or even basic effectiveness of this kind of approach.

In summary, the creation of performance models and time-series models is based on concrete historical data. Thereby one models is usually specific to one application, and even when the workload is different, a specific model is required[17, 43, 71, 76]. The training

process requires considerable extra online or offline efforts, especially for an accurate model. Otherwise, the prediction accuracy may be unacceptable for applications with strict SLO constraints. Runtime monitoring may be needed for model updates or feedback-based error correction.

Feedback-based Approaches

In contrast to prediction-based approaches, feedback-based resource allocation measures the system performance online and adjusts the allocation based on the performance measurements. A feedback-based allocation system has three important aspects: the performance metrics to monitor, the decision mechanism, and the action to change resource allocations. For CPU resource management, the action is changing the CPU allocation. The decision mechanism still requires a model that describes the relationship between performance and CPU resource allocation. Nevertheless, the model is usually not built using machine learning techniques.

Control theory has been used in the literature to regulate resource allocation to meet the target performance of applications. Padala et al. propose such a control system that automatically adapts to dynamic workload changes to achieve application SLOs [58]. The relationship between application performance and resource allocation cannot be represented by a simple linear model since application behaviours are non-linear and workload-dependent. In the paper, it is assumed that the application in the neighbourhood of an operating point can be approximated by a linear model. For every control interval, an online model estimator re-computes a linear model that approximates the quantitative relationship between resource allocations and normalized application performance using the auto-regressive-moving-average (ARMA) filter. With the model, an optimizer determines the resources allocated to an application by minimizing a cost function which includes both the performance cost and control cost. Performance cost refers to a penalty for the deviation of the measured application performance from the SLO. Control cost is higher when the controller makes a larger change in the resource allocation in an interval. Moreover, this system has a NodeController that allocates resources among applications based on the resource request made by each optimizer. When the available resources are insufficient, this NodeController picks an allocation that locally minimizes the error between the resulting normalized application performance and its target value.

Kalyvianaki et al. design controllers to provision a multi-tier web application with CPU resources to adequately serve incoming requests from a varying number of clients [36]. The application is first run offline with the total CPU capacity to derive a model that describes the relationship between the CPU utilization and the allocation that maintains

the target performance. In this model, the allocation is the mean utilization augmented by an additional extra value. When the model is ready, the Kalman filter technique is used to design three feedback controllers to adjust CPU allocation based on the measured utilization. A basic Single-Input-Single-Output (SISO) controller is designed for a single component in the application. The basic controller is extended to a Multiple-Input-Multiple-Output (MIMO) controller that allocates CPU resources between components. With this controller, the allocation for each component is adjusted based on the error of the current component plus the errors caused by other components. Only stationary process and measurement noises are considered in the two controllers. The third MIMO controller is designed to adapt resource allocation to unpredictable time-varying workloads by considering non-stationary noise. Utilization is easily measured at the server side with negligible overhead and widely applicable across applications. This work does not investigate the relationship between utilization and application performance, but uses a linear model between allocation and utilization.

A self-aware programming framework named SEEC is proposed in [34]. SEEC employs an adaptive control system and can be applied to resource management. The SEEC framework clearly defines three phases: observation, decision, and action. In the observation phase, applications need to define their performance metrics to monitor as 'heartbeats' and express their performance goals in terms of their heartbeats [33]. Applications need to be modified to support the heartbeat measurement. In the action phase, system programmers can define a set of actions with benefits and costs through a SEEC API. The action for each control interval is determined by the decision mechanism. Changing the CPU allocation can be a type of action. The decision phase has three adaptation levels. Level 0 controls the action applied to the application to reduce the error between the heartbeat goal and the observed heartbeat via a basic classical controller. Level 1 extends the basic control system with an adaptive control system which estimates the application workload online. Level 2 models system behaviours and fixes any errors about the input benefits and costs of actions. The three different controllers have multiple parameters. It is not clear whether these parameters depend on the type of target application.

Fuzzy control shows a strong capability of incorporating domain knowledge to deal with the inherent nonlinearities of computing systems [30]. For instance, a resource management framework based on feedback fuzzy control, FC2Q is presented in [2]. The purpose of FC2Q is allocating the minimum CPU resources to applications running on a physical server without violating their SLOs. Therefore, this system monitors application performance and compares the measurements with user-specified SLOs. FC2Q does not use a model of system behaviours, but bases the fuzzy controller on a general relationship between CPU utilization and application performance. The rules about when and how to change the

Table 2.1: FC2Q Control Rules

	full utilization	high utilization	low utilization
SLO violated	large increase	small increase	no change
SLO met (far)	small increase	small decrease	large decrease
SLO met (near)	small increase	no change	small decrease

CPU allocation in the fuzzy controller are shown in Table 2.1. In the implementation, the CPU allocation is changed by updating the cap value in the Xen credit scheduler. However, FC2Q does not find a practical way to measure application performance at the server side. In its evaluation, FC2Q gathers performance measurements from a specific application by interacting with the workload driver used to generate its workload.

Lo et al. present a dynamic control system named Heracles using intuitive rules rather than the control theory [51]. This system ensures that latency-critical (LC) tasks meet their SLOs and maximizes the resources allocated to best-effort (BE) tasks. Heracles dynamically adjusts resource allocation based on the difference between the measured latency and the SLO latency (called latency slack). Similar to the FC2Q system discussed above, a SLO latency is needed as the reference input in Heracles. The system is organized as three subcontrollers (cores & memory, power, and network traffic) coordinated by a top-level controller. The top-level controller disables and enables BE tasks based on the latency slack of the LC workload, and instructs subcontractors to change the resources allocated to BE tasks. Heracles uses a single subcontroller to manage CPU and memory. The first constraint for the subcontroller is to avoid memory bandwidth saturation. When the top-level controller signals more resources for the BE workload and there is no DRAM bandwidth saturation, the subcontroller uses gradient descent to find the maximum number of cores that can be allocated to BE tasks. Every time the subcontroller increases the CPU allocation to the BE workload, it checks DRAM bandwidth saturation and SLO violations for the LC workload. The cpuset subsystem of Linux cgroups is used in Heracles to enforce CPU allocation. The controller measures the actual task latencies from the client side. Heracles is not a general-purpose resource manager. It is designed for this specific scenario for tasks with short latencies and certain best-effort workloads.

Arachne is another system based on intuitive rules [62]. It is a core-aware user-level thread management system and contains a feedback-based core arbiter to allocate CPU cores among applications. The core arbiter uses the cpuset system of Linux cgroups to allocate specific cores to specific applications. The core arbiter adjusts the core allocations as application requirements change. The advantage of Arachne is that its core arbiter determines CPU allocation based on two application-agnostic metrics. The system requests

one more core for an application when the average load factor across all cores allocated to that application reaches a threshold. Here the load factor is the average number of runnable user threads on a core. When the CPU utilization is less than the allocation by some extent, the system reduces an application's requested number of cores by one. Utilization is not used for scaling up because the best utilization for scaling up depends on the burstiness and overall volume of the workload. The core arbiter collects requests from each application about how many cores it needs and uses a simple priority mechanism to divide the available cores among competing applications.

Besides CPU allocation, resource managers can vary CPU states to pursue power efficiency. A feedback-based algorithm, POLARIS, is designed to control CPU frequency scaling for transactional database systems in [41]. Each workload known to POLARIS is associated with a latency target. The objective of this algorithm is to minimize CPU power consumption while ensuring that each transaction is completed within the latency target. POLARIS estimates the queueing time and execution time of a transaction at a certain CPU frequency, and then adjusts the CPU frequency correspondingly. POLARIS tracks the 95th percentile of measured execution times over a sliding window of the most recent transactions that run at a given frequency. The tracked value is used as the estimated execution time at this frequency. The measurement of transaction latency needs modification in the database systems. The methodology in this thesis can also be used for power management, yet this direction is not explored in this thesis.

In summary, most existing feedback-based systems for multi-tenancy resource allocation require some level of integration with the target system. Some proposals rely on application- or workload-specific performance modelling in their control system [36, 58]. Some systems require modifications in applications to measure application performance as feedback [34, 41, 58]. Because it is difficult to measure application performance at the server side, some papers escape this problem by measuring application performance at the client side [2, 51]. Thereby, application-agnostic metrics such as CPU utilization are considered [62].

The CPU resource manager in this thesis uses an application- and workload-agnostic metric to measure application performance, and does not require explicit performance objectives for each application. It does not require offline training, but performs online measurements and feedback control to allocate resources with substantially less complexity than the systems described above.

2.2 Background

This section presents two experimental studies that investigate how existing CPU sharing mechanisms perform. These studies provide the background for the work on CPU management in this thesis. First, when multiple CPU cores are available, CPU resources can be allocated to a workload in two different ways, which are referred to as *partitioning* and *time-based allocation* in this thesis. The first study explores the impact of the two approaches on application performance. Second, a CPU sharing mechanism determines a theoretical share of CPU resources that each workload can consume. When workloads compete with each other for CPU resources, an effective CPU sharing mechanism should guarantee that each workload can obtain its theoretical share of CPU resources. The second study investigates whether existing CPU sharing mechanisms can enforce such resource isolation.

As introduced in Section 2.1, many current resource managers choose Linux cgroups for resource sharing [26, 31, 51, 62, 70, 72], while some other systems like Xen [10] use mechanisms similar to Linux cgroups. Therefore this section analyzes the performance of Linux cgroups as a typical case.

2.2.1 Partitioning and Time-based Allocation

This section compares the two different ways to allocate CPU cores. In this thesis, the term *time-based allocation* is used to refer to the traditional way of sharing CPU cores among multiple threads using time multiplexing. Time-based allocation considers CPU resources as time slices, but does not restrict threads to specific cores. For a single-core CPU, this is the only option to build a multiprocessing system at all. However, modern servers have many cores at their disposal, which opens a new direction for CPU provisioning. The availability of multiple cores allows partitioning the set of cores rather than sharing all the cores. This alternative option is referred to as *partitioning* in this work. Partitioning exclusively allocates certain cores to different groups of threads.

When multiple cores are available, CPU resources should not simply be considered as time slices. For complex parallel software systems, e.g., database systems, the number of cores determines the number of threads running simultaneously. Too many threads from the same system running at the same time may increase contention in the software system. Since a time-based allocation mechanism does not restrict which CPU cores can be used simultaneously by an application, the maximum parallelism (i.e., the number of threads running simultaneously on multiple cores) of an application is not restricted. Partitioning

limits the number of cores available to each application, thus largely avoiding the drawbacks associated with the excessive parallelism.

As introduced in Section 2.1, the *cpuset* and *cpu* subsystems of Linux cgroups are used in practice for partitioning and time-based allocation respectively. The cpuset subsystem specifies the physical cores allocated to a cgroup. This work uses only exclusive allocation in the cpuset subsystem to implement partitioning. The cpu subsystem provides two time-based allocation mechanisms (quota and weight). The quota mechanism limits the maximum amount of CPU time that a cgroup can consume during a reference period. The weight mechanism determines the fraction of CPU time each cgroup can obtain based on relative weights of the cgroups. The two subsystems of Linux cgroups are used to conduct a series of experiments to verify drawback of excessive parallelism.

The testbed machine has 4 AMD Opteron processors (4x16 cores) and 512 GB memory. The operating system is Ubuntu Server 16.04 with Linux kernel 4.4.0-137-generic. Applications use memory interleaved across NUMA nodes on the sockets. Thus the memory interference is balanced for workloads. In the following experiments in this section, each case is executed 20 times. Only the average results are shown in figures because the coefficients of variation for the result sets are all less than 0.04.

The first experiment uses a representative transactional database workload with the MariaDB system [23] and transactions from the TPC-E benchmark [12]. The total size of the database is about 8 GB. To reduce the impact of disk I/O, the buffer pool size is set to 16 GB and the innodb log file size is increased to 4 GB. The benchmark program has 50 client threads, and each thread submits a new transaction after the previous transaction is completed. In the database experiments, the database server can use up to 32 cores exclusively. Another 16 cores on the same machine are used for clients to generate requests. Clients connect to the database server via the local network interface.

The stable TPC-E workload is repeated with core partitioning and the quota mechanism to facilitate a comparison. With partitioning, the numbers of physical cores is varied from 1 to 32. Figure 2.1 shows the performance curve of this workload. Increasing the number of CPU cores initially increases the throughput of the database system. However, if too many cores are used, the throughput does not increase anymore but slightly declines. The number of CPU cores determines the number of threads running simultaneously. When the number of cores is greater than the ideal level of parallelism (18 in this case), contention can make a parallel system less efficient, because CPU cycles are spent on solving contention among threads without corresponding work progress. Therefore, CPU cycles are wasted while the application performance does not improve.

When CPU allocation is changed via the quota mechanism, the database server can

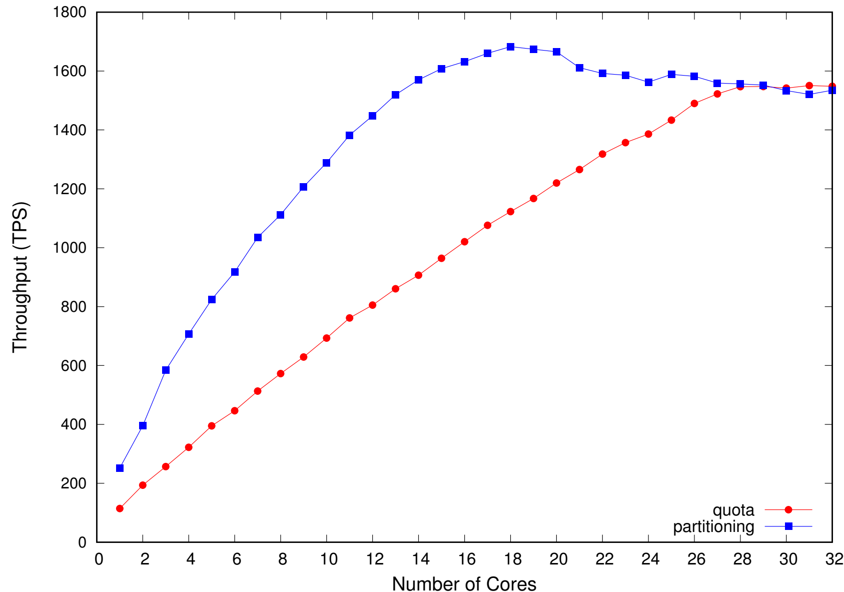


Figure 2.1: Performance Difference of TPC-E Workload with Partitioning and Quota

access all 32 physical cores. The quota time is varied, equivalent to 1 to 32 effective cores. For simplicity, "N effective cores" is used to represent the CPU time which is equivalent to N CPU cores during a period of time in this thesis. As Figure 2.1 indicates, the system performance is quite different. With the same number of cores, the performance with the quota mechanism is worse than that with partitioning in most cases. With the quota mechanism, the potential system parallelism is 32, and the performance of a saturated system increases with the quota time. The best throughput with the quota mechanism is achieved when all the CPU time is allocated to this workload. Yet 32 is not the ideal level of parallelism for this database workload, so even the best result with a quota of 32 effective cores is worse than the result with a partitioning of 18 physical cores.

Additional metrics about the TPC-E database workload are measured to analyze the parallelism issue. Figure 2.2 shows the overall performance of this workload, measured as transaction throughput, in comparison to the number of CPU cycles that are spent processing the workload. As the number of cores increases via partitioning, the number of CPU cycles keeps increasing linearly, independently of the stalled and declining transaction throughput. This result demonstrates that CPU cycles are wasted when parallelism exceeds the ideal level. An efficiency loss of this kind can be an issue for time-based allocation mechanisms. When running across a large number of cores, an application may execute with excessive parallelism.

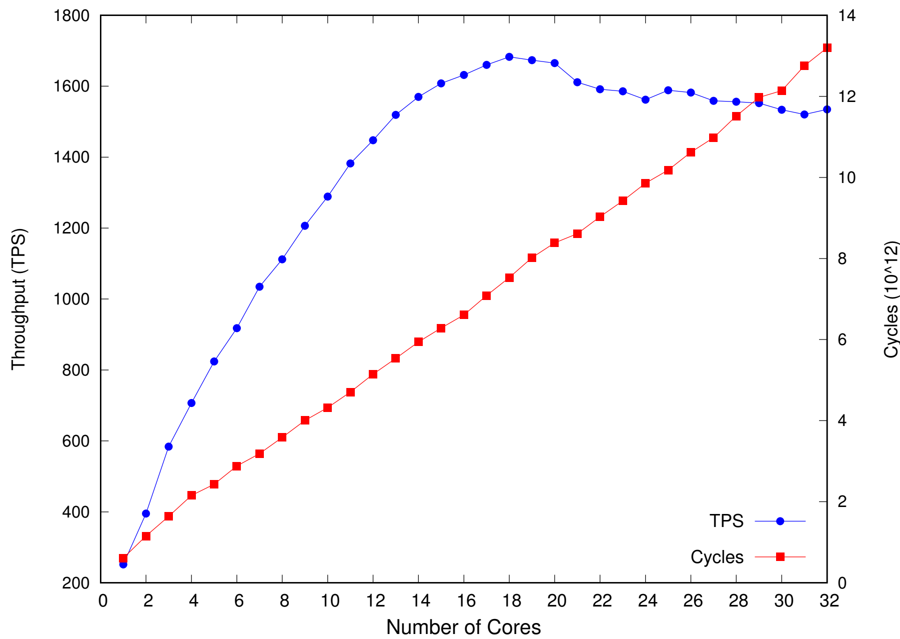


Figure 2.2: Cycle Usage with TPC-E Workload

A significant fraction of the inefficiency is caused by resource arbitration and contended synchronization primitives. For instance, multiple threads can use a spin lock or a blocking lock for synchronization. With a spin lock, a thread spins on looping memory reads, and continuously utilizes CPU resources. Similarly, the primitives in lock-free programming are usually performed in a loop to make repeated attempts, and thus the CPU is kept busy. The MariaDB server used in the experiments above uses spin locks for synchronization and atomic operations in its b-tree search and log writing functions. Contemporary processors use atomic machine instructions to facilitate synchronization. For example, this kind of atomic instruction is characterized by the LOCK prefix on AMD/Intel processors. Using hardware performance monitoring units, it is possible to count both the number of atomic instructions retired, as well as the CPU cycles spent on these instructions. The corresponding measurements during the experiment presented above are also collected. The results are shown in Figure 2.3. Similar to the previous observation, the growth of the number of atomic instructions gradually slows down with increasing parallelism, while the number of cycles spent on atomic instructions increases linearly with the number of cores. When more than 20 cores are allocated to the database server, for each core added, about 20% - 50% of the increased cycles are spent on atomic instructions.

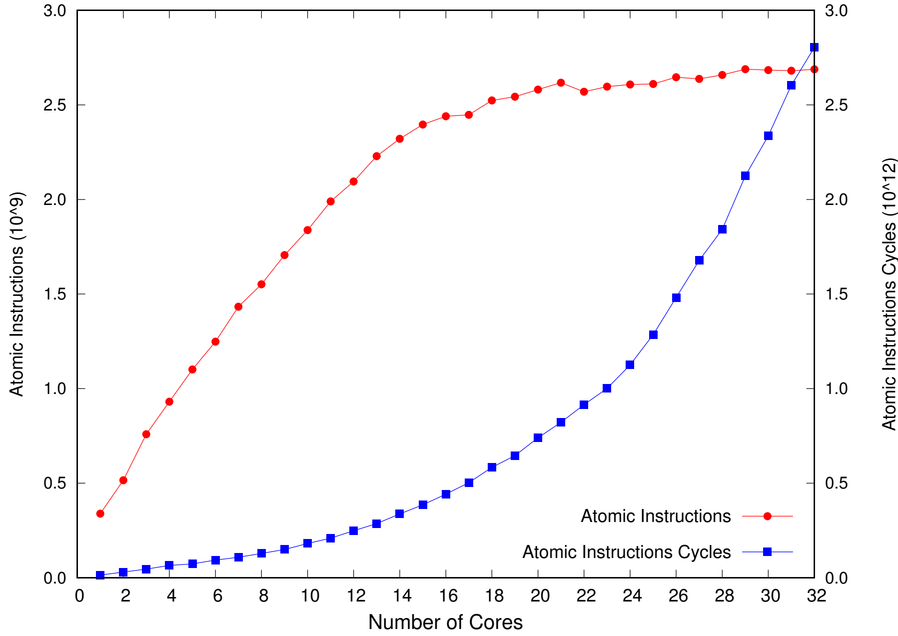
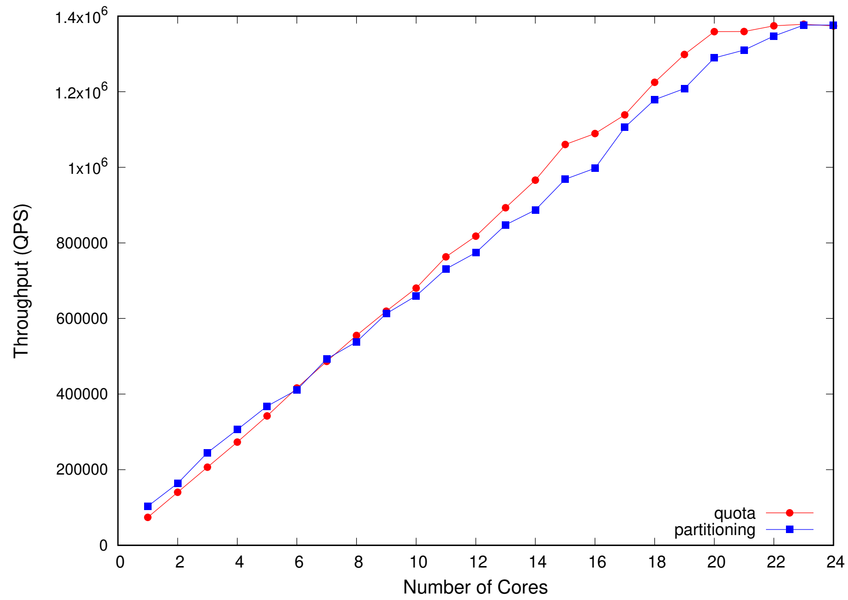


Figure 2.3: Atomic Instruction Overhead with TPC-E Workload

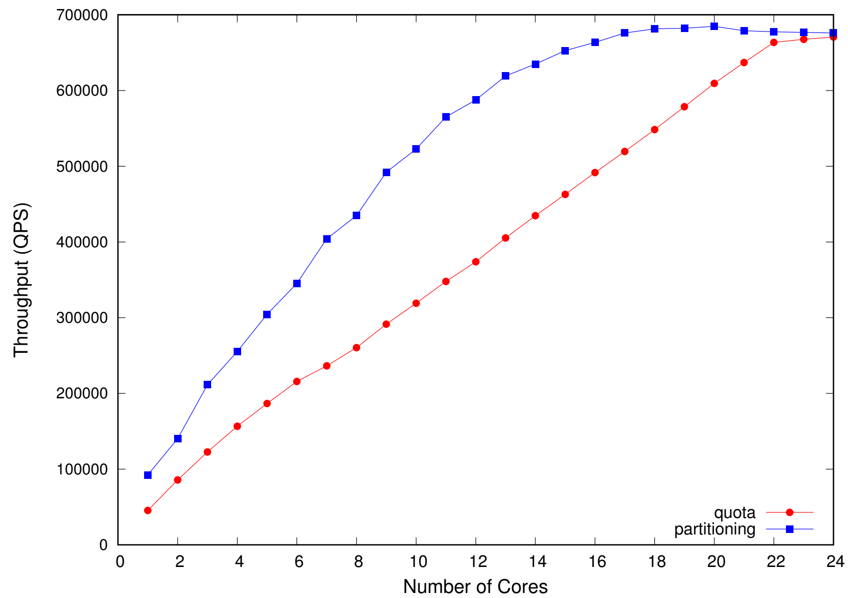
A similar experiment is conducted with an in-memory key-value store Memcached [52] to confirm the impact of parallelism. The Memcached server starts with 24 threads and 50 GB memory, and 24 CPU cores are used by this server exclusively. A load generator Mutilate [53] is used to recreate the Facebook "ETC" request stream from [4]. Two different workloads are generated by varying the number of records and the ratio of updates. A normal caching workload has 1,000,000 records and only 10% of the operations are updates. In contrast, another synthetic workload has 500,000 records and 50% of the operations are updates. The conflicts among operations are increased on purpose to study the impact of parallelism.

Figure 2.4(a) shows the queries per second (QPS) of the normal caching workload with core partitioning and the quota mechanism. Because parallelism has negligible impact on this read-heavy workload, the throughput keeps increasing with more CPU resources regardless of the allocation method. In contrast, Figure 2.4(b) shows that for the write-heavy workload, core partitioning and the quota mechanism result in different performance even with the same number of cores. When core partitioning is used, the throughput of the server first increases with the number of physical cores and then becomes steady.

The throughput results in Figure 2.4(b) indicate that the excessive parallelism impact



(a) Throughput of Typical Memcached Workload



(b) Throughput of Write-Heavy Memcached Workload

Figure 2.4: Memcached Performance with Quota and Partitioning

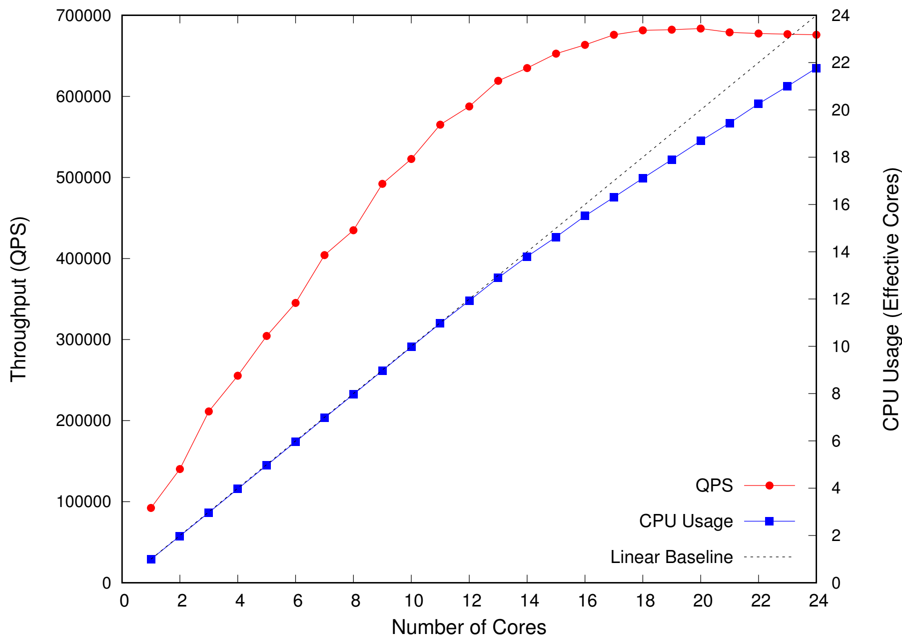


Figure 2.5: CPU Utilization with Write-heavy Memcached Workload

the efficiency of the Memcached server when there is contention among threads in the write-heavy workload. The Memcached server does not use as many atomic operations as MariaDB, but it uses blocking locks for synchronization. The blocking synchronization prevents the Memcached server from fully utilizing CPU cores and improving the throughput.

With a blocking lock, a thread releases its CPU, puts itself in a waiting queue and blocks. Hence serious thread contention reduces CPU usage. Figure 2.5 shows the CPU usage of the write-heavy Memcached workload with a varying number of physical cores. The number of allocated cores is plotted in this figure as a baseline. With fewer than 12 cores, the Memcached server is very efficient and almost fully utilizes all the allocated cores. However, the CPU usage of this workload does not increase linearly. The difference between the utilized CPU resources and the available CPU resources becomes larger when the parallelism increases. Threads in the Memcached server yield more CPU resources as the thread contention intensifies.

Moreover, blocking locks also result in extra costs of CPU resources. Based on the profiling result of the write-heavy workload, the functions for blocking synchronization in Memcached consume a higher ratio of cycles when more physical cores are allocated via par-

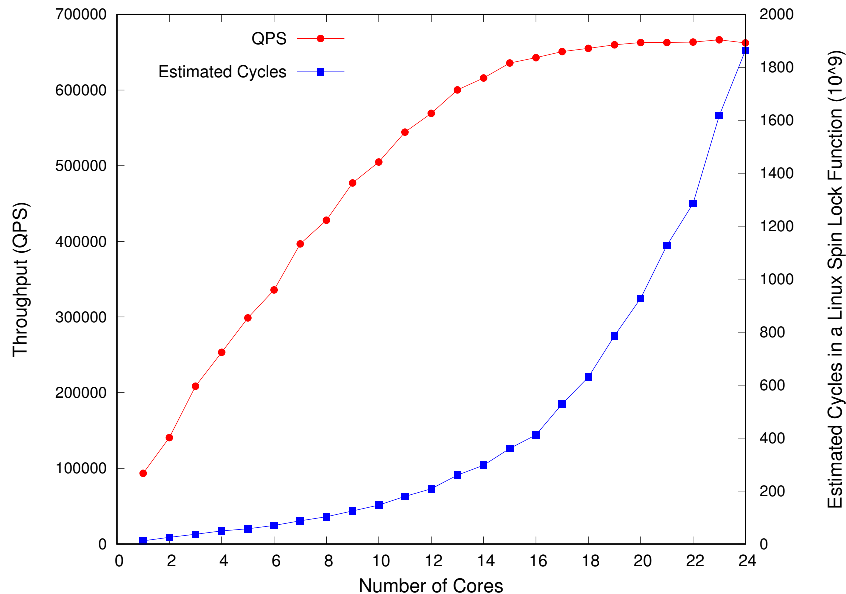


Figure 2.6: Estimated Cycles in a Linux Spin Lock Function with Write-heavy Memcached Workload

titioning. A typical example is a kernel function named *native_queued_spin_lock_slowpath*, which is used for a blocking lock called futex in Linux. While with a single core, this function uses less than 2% CPU cycles, with 24 cores, it uses more than 15% of cycles and becomes the function using the most cycles. The percentage of cycle samples spent in each function is measured by the OProfile tool [56]. Figure 2.6 presents the estimated cycles used by this function with different number of cores calculated from the measured total cycles multiplying the profiled percentage. It can be observed that with an increasing number of cores, the number of cycles used by this function increases in a super-linear manner.

In the last experiment, the difference between the weight mechanism and partitioning is investigated. The weight mechanism is also a time-based allocation mechanism. This experiment confirms that the parallelism issue does not only exist in the quota mechanism. The weight mechanism is designed to share CPU resources among multiple workloads instead of limiting the usage of a single workload, so two identical TPC-E workloads are used in this experiment. The total number of cores is the same for the two sharing methods. With partitioning, each workload can obtain only half of the cores for database servers. With the weight mechanism, each workload can access all cores but the workloads compete with each other equally. The CFS scheduler schedules threads to cores. This experiment

is repeated with a total of 32, 16 and 8 server cores respectively.

Figure 2.7 presents the average throughput of the two identical workloads with partitioning and the weight mechanism respectively. With either mechanism, both workloads consume about half of the CPU resources and achieve similar performance. However, the figure clearly shows that partitioning delivers better application performance than the weight mechanism. The performance improvement with partitioning is 45% for 32 cores, 18% for 16 cores, and 9% for the 8-core case. This symmetric experiment illustrates the impact of parallelism on application performance when time-based allocation is used for CPU sharing.

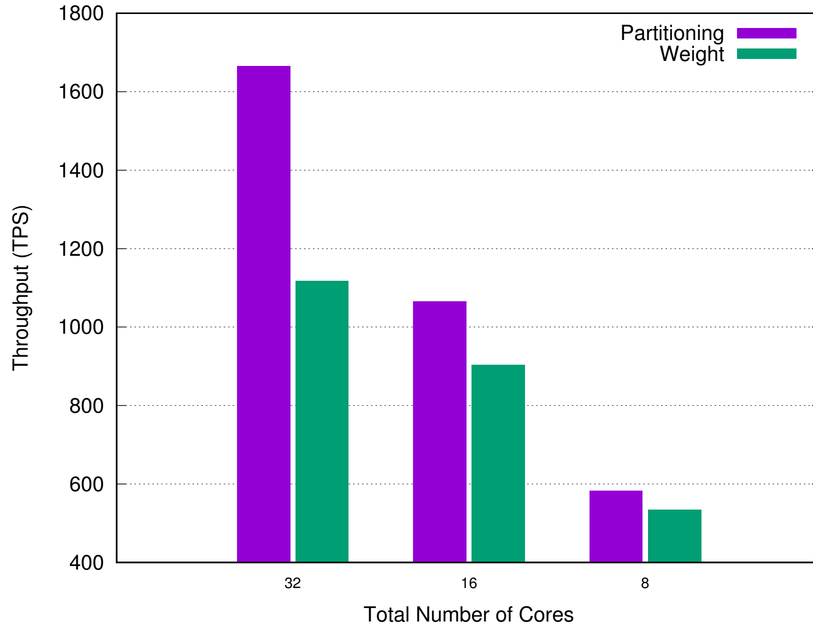


Figure 2.7: DBMS Performance with Weight vs. Partitioning

2.2.2 Resource Isolation

This section examines whether the two time-based allocation mechanisms enforce resource isolation in typical proportional sharing scenarios. The weight mechanism in Linux cgroups is designed for proportional sharing. In a real-world resource manager, this mechanism is also used to emulate multiple priority levels [26]. The quota mechanism is designed to address the isolation problem of the weight mechanism [67]. The quota mechanism can

also support proportional sharing. Given the total number of CPU cores, the quota of each cgroup can be calculated based on their weights. For example, with a total of 16 cores and a 3:1 weight ratio, if the quota for one workload is equivalent to 12 cores, then a minimum of 4 cores would be available for the other workload.

This experimental study chooses two different multi-threaded programs. First, a multi-threaded program named cpuhog is developed to represent CPU-intensive workloads. This program keeps CPUs busy doing computations. The second program is the MariaDB database server and the TPC-E workload in the previous study is used. To establish the experiment conditions, the workloads are tuned to require more CPU resources than the quantity allocated to them, thereby creating CPU resource contention between workloads. A total of 16 physical cores are used by the workloads. The cpuhog program uses the same number of threads as cores. The database server has a fixed number of 50 threads to handle requests. The database client uses other separate cores. When the cpuhog program runs in isolation with 16 cores, it fully utilizes all the cores. When the transactional database workload is executed in isolation with 16 cores, its CPU usage is 13.48 cores by average.

In the experiments, the time for each run is 180 seconds. CPU usage of each workload is measured using the Linux perf_event interface [73]. The measured CPU time is converted into a number of CPU cores in the results. Every result shown in this section is based on 20 samples, and the coefficient of variation of samples is less than 0.05 if not specified.

In the first experiment, 16 cores are shared by two identical cpuhog workloads. Different weight ratios are assigned to the two workloads. The CPU usage results of the cpuhog workload with the lower weight are shown in Figure 2.8. The weight ratios are also shown in the figure. In addition, the theoretical CPU allocation to the lower-weight workload is depicted as the ideal baseline. What can be clearly seen from this figure is that the quota mechanism can effectively enforce resource isolation. With the weight mechanism, the CPU usage of the two cpuhog workload basically matches their theoretical allocation, but there are aberrations from the ideal values in certain cases.

In the second experiment, two identical TPC-E database workloads are executed simultaneously with 16 cores. Figure 2.9 shows the CPU usage results of the lower-weight database workload with different weight ratios. With the weight mechanism, the lower-weight database workload uses more CPU resources than the amount it should obtain except in the equal weight scenario. Hence, the share of CPU resources for the higher-weight workload cannot be guaranteed even when the same workloads are running simultaneously. Such a result is due to how the underlying Linux scheduler manages CPU time. The Linux CFS scheduler allocates time slices to each thread based on their weights. Threads in the database server may yield the CPU before using up a time slice. Their unused CPU time

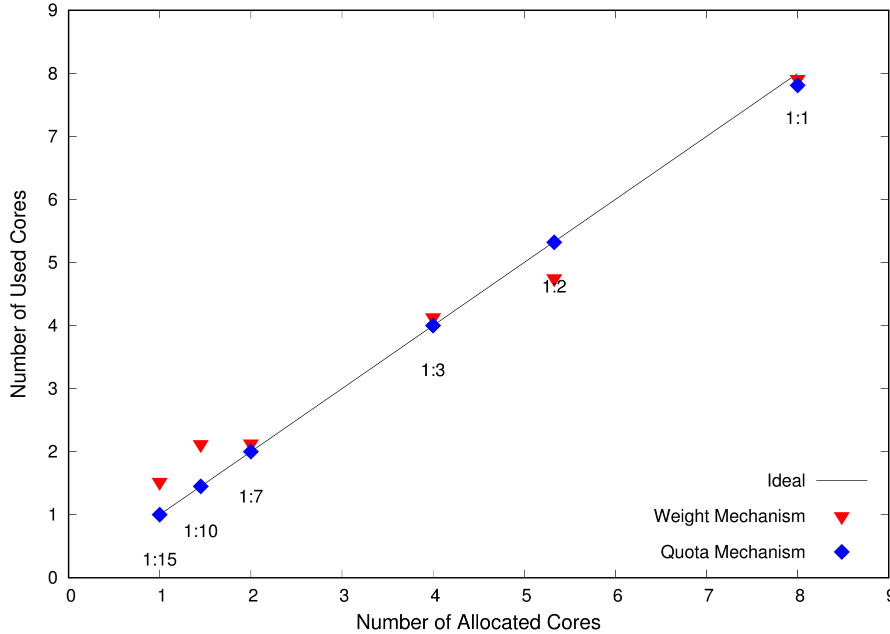


Figure 2.8: CPU Usage of the CPUhog Workload with the Lower Weight

is not carried over to the next time slice (see Section 2.1.1). The two database workloads in this experiment yield CPU resources to each other. With Linux CFS, the higher-weight workload has more threads running while the lower-weight workload has more threads waiting. Therefore, the lower-weight workload obtains extra time compared to its theoretical allocation. As an exception, when the two database workloads have the same weight, their CPU usage is similar. Neither of the workloads uses up to 8 cores. The TPC-E workload cannot fully utilize 8 cores due to serious thread contention. Similar to the previous experiment, the quota mechanism restricts the CPU usage of the lower-weight workload, and thus guarantees that the higher-weight workload can obtain its theoretical share.

A combination of two different workloads is tested in the third experiment. Specifically, the database workload and the cpuhog workload are executed at the same time with different weight ratios. This experiment chooses an extreme weight ratio of 2:10240, which is used in the container management system Kubernetes [26]. Kubernetes uses the cpu subsystem of Linux cgroups to control CPU allocation among containers with different priorities. Kubernetes has three priority classes: 'Guaranteed', 'Burstable' and 'BestEffort'. A 'BestEffort' workload does not have any quality of service guarantee, and should give up resources to other workloads when other workloads have requirements. 'Guaran-

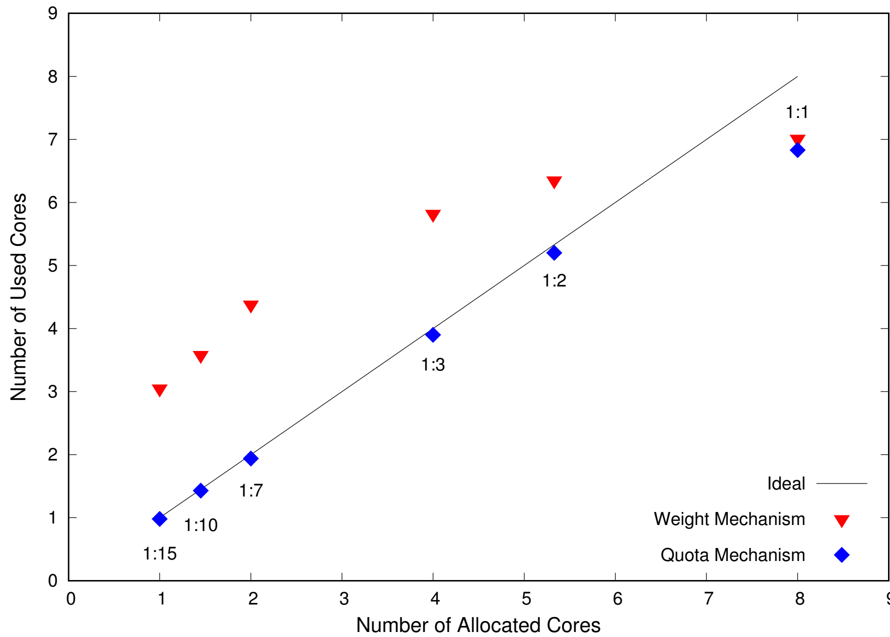


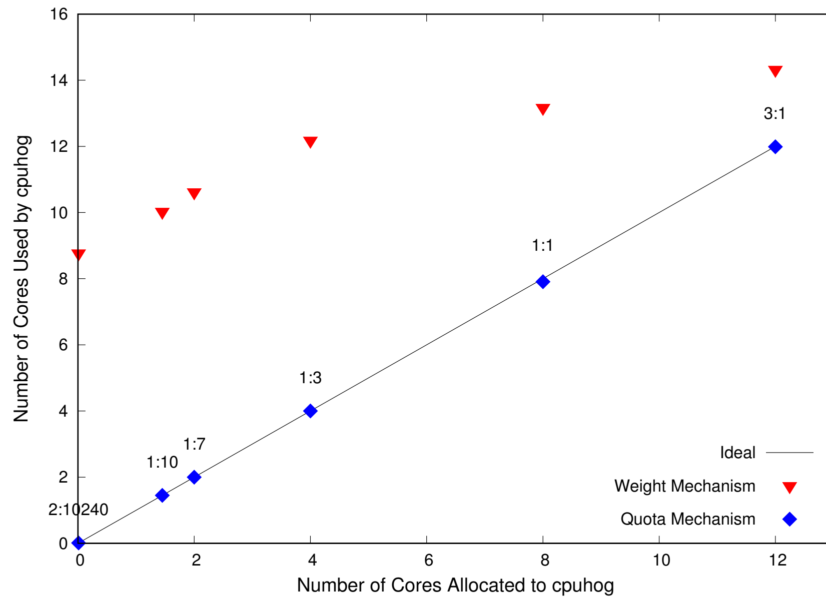
Figure 2.9: CPU Usage of the TPC-E Workload with the Lower Weight

teed' and 'Burstable' workloads are assigned 10240 shares of CPU time while 'BestEffort' workloads can obtain only 2 shares.

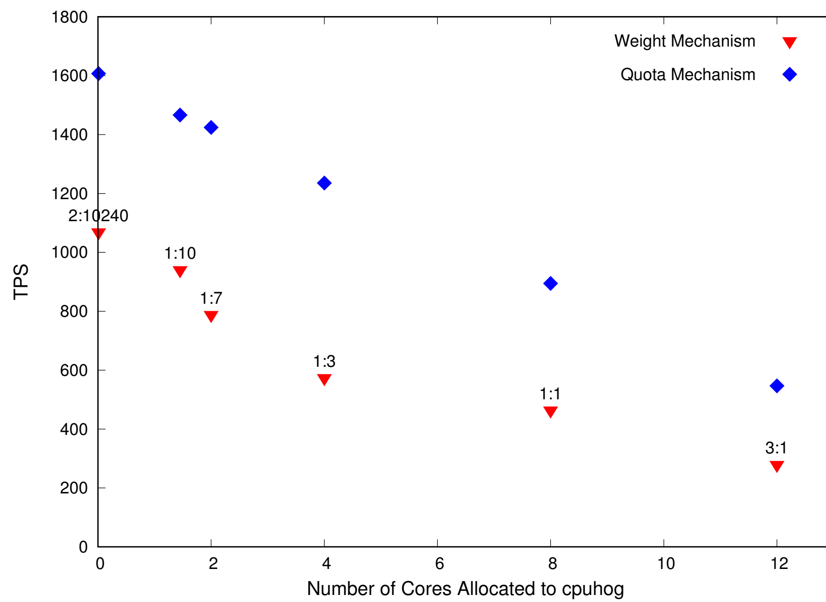
The CPU usage results of the cpuhog workload with different configurations are shown in Figure 2.10(a). What stands out in this figure is that with the weight mechanism, the cpuhog workload can consume more CPU resources than its theoretical allocation. The cpuhog program gets more chances to run when the database server yields the CPU. Threads in cpuhog are CPU-intensive and always use up the time slice allocated to them. Even an extreme weight configuration like 2:10240 cannot suppress the cpuhog program.

Figure 2.10(b) shows the throughput of the database server with the two mechanisms in the corresponding experiments. The throughput with the weight mechanism is lower than that with the quota mechanism, because the weight mechanism fails to provide the proper share of CPU resources to this workload. In contrast, the quota mechanism enforces the resource isolation based on the configured weights. Even when the database workload does not use all its quota CPU time because of thread contention, the neighbour cpuhog program cannot use more idle resources than its quota.

Taken together, the results of the three experiments show that the weight mechanism in Linux cgroups cannot consistently enforce CPU resource isolation. Workloads do not



(a) CPU Usage of the CPUhog Workload



(b) Throughput Results of the TPC-E Workload

Figure 2.10: Results with Different Workloads

always obtain their theoretical shares of CPU resources based on their weights. Workloads with lower weights, especially CPU-intensive workloads, may be allowed to filch CPU time from other workloads.

Most existing proposals for resource sharing among multiple tenants compute CPU allocation as a fraction of the overall CPU capacity and share all the available physical cores via the weight mechanism [16, 58, 76]. The main benefit is that general purpose OS schedulers are typically work-conserving. Thus, idle resources are automatically available for other workloads. Although the static quota mechanism can enforce resource isolation, it is not work-conserving. Given the dynamic nature of real-world workloads, a static quota configuration will not fit all the time. When the CPU requirement of one workload shrinks and some CPU resources become available, other workloads are still not allowed to cross the quota limit and utilize the idle CPU resources. Statically partitioning a set of cores is not work-conserving either. Partitioning does not allow applications to access each other's cores. The resource isolation is thus effectively enforced in this way. If quota values or core partitioning were adaptive to workload requirements, idle CPU resources could be utilized by other workloads, and thus both resource isolation and work conservation could be achieved.

Chapter 3

Metrics for Online Performance Measurement

A feedback-based solution is investigated in this thesis to address the problem of adaptive resource allocation. A feedback-based system is driven by changes in application performance that happen in response to changes in resource allocation. An online performance metric is required by such a feedback-based system as the feedback signal. Two features are beneficial for a feedback-based resource management system. First, different types of applications should be supported at the same time. Second, measurements and adjustments should be made in a timely manner to handle workload dynamics.

This chapter first discusses metrics used in the literature, and then introduces a new generic metric for online performance measurement. The new metric is application-agnostic and presents a uniform performance assessment for different types of applications. Measurements for this metric are non-invasive and can be obtained using hardware performance monitoring units (PMUs) without modifying the OS or application software. Meaningful data can be collected in a short time interval, which permits quick adjustments to CPU allocations.

3.1 Existing Performance Metrics

3.1.1 Application-Level Metrics

Application-level metrics can show performance changes and facilitate feedback-based algorithms. Previous studies in the field of adaptive resource allocation have used specific application-level metrics as feedback, or calculated service level agreement (SLA) values based on these specific metrics [3, 36, 51, 62, 76]. However, different applications may have different quantitative metrics to measure their performance of interest.

One application's performance metric may not be suitable for other applications. Some applications' performance is measured using throughput, some applications are latency-sensitive, and some care about both. For example, the transactions per second (TPS) metric is usually used to judge the performance of an online transaction processing (OLTP) database system. However, TPS is not suitable for performance measurement of an online analytical processing (OLAP) system. The performance metric of interest for an OLAP system is typically query execution time instead of throughput. Analytical queries can have widely varying execution times, from seconds to hours. Therefore, queries per second or queries per minute cannot reflect the real-time progress of an analytical workload. When a system needs to run various applications, the resource manager has to support all the different application-level performance metrics.

Moreover, the length of the measurement interval determines how fast a feedback-based management system can make adjustments. When using performance measurements as feedback, a resource manager requires each measurement to have a short interval. For instance, TPS can be measured over a short period of time, so a resource manager can take several measurements quickly and compare performance changes of a database server. Conversely, if an application-level metric needs to be measured over a long period of time, e.g, queries per hour for an OLAP database workload, the resource manager can only make an adjustment after a long time. Hence this kind of metric is not suitable for timely feedback measurement. Taking the measurement interval into account, certain workloads, e.g., OLAP database workloads, do not have a suitable application-level metric for online performance measurement at all.

Another drawback of application-level metrics is that their measurement typically requires instrumentation in applications. The instrumentation may need modification in the application source code and introduce extra costs to the execution of workloads. For example, to measure the throughput of a database system, the number of transactions completed across all the connections must be recorded inside the database system. In this way, the measurement of application-level metrics can be costly in practice.

3.1.2 CPU Usage

Certain application-agnostic metrics for resource management have been suggested in the literature [36, 62, 78]. This section considers an important application-agnostic metric, CPU usage. CPU usage is the total CPU time used by a program during a period. This metric is usually converted to another format by comparing to the elapsed CPU time on a CPU core during the same period. For example, in the Linux perf tool, the usage is converted to the number of effective cores [73], while in the default mode of the Linux top tool, the usage is presented as a percentage which may be greater than 100% [24].

The advantage of CPU usage is that this metric is not specific to any application. When CPU usage is used in an allocation algorithm, no application information is required. Moreover, it is easy to measure CPU usage because operating systems already provide necessary measurements. For example, Linux records the statistics about CPU time used by each individual process in the /proc/[pid]/stat file.

However, CPU usage cannot accurately show how application performance changes. In a multi-threaded software system, multiple threads are coordinated via synchronization primitives. The resulting instructions spend CPU time, but do not directly process the workload. The experiment with the write-heavy Memcached workload in Section 2.2.1 shows such an example. The CPU usage continues to increase regardless of whether the application performance increases or remains unchanged (see Figure 2.5 on Page 25). Moreover, when the allocated CPU resources is reduced, a workload's CPU usage may also decrease. It is impossible to judge whether the CPU requirement of a workload decreases based on only CPU usage under this circumstance.

This work also investigates CPU utilization of an application. CPU utilization does not have a clear definition and may be calculated in different ways for different purposes. In this thesis, CPU utilization refers to the ratio of the CPU time used by an application and the total CPU time available to this application. This metric compares CPU usage to allocated CPU resources. Some previous resource management systems employ a similar utilization metric as the feedback to control CPU allocation [15, 36, 45, 62].

CPU utilization is also application-agnostic. Only measurements of CPU usage are required to calculate this metric. Moreover, from the perspective of resources, when the CPU utilization of an application is low, this application does not require all the allocated CPU time to process its workload. Accordingly, it is reasonable to reduce the CPU resources allocated to this application. However, similar to CPU usage, CPU utilization does not accurately reflect application performance. In the experiment mentioned above (see Figure 2.5 on Page 25), when more than 14 cores are allocated to the Memcached server,

the difference between the used CPU time and the available CPU time increases, i.e., the CPU utilization declines regardless of whether the application performance changes or not. Therefore, even for a stable workload, it is difficult to determine whether to increase the CPU allocation and when to stop allocating more resources to a workload. A previous paper indicates that CPU utilization is inaccurate to reflect application performance and guarantee that the SLO of an application is met [50].

For a dynamic workload, CPU utilization can indicate when the CPU requirement of a workload decreases, since the CPU utilization of a workload decreases as a consequence of the decrease in that workload's CPU requirement. Nevertheless, with a high CPU utilization, even if the requirement of a workload increases further, the CPU utilization will not change dramatically. Thereby, this metric cannot always indicate whether the CPU requirement of a workload increases when the CPU allocation does not change.

3.2 User-level Instructions Per Time

This section proposes a new metric *user-level instructions per time* (UIPT) as a real-time indicator of system performance. UIPT is calculated as

$$\frac{User\text{-}level\ Retired\ Instructions}{Measurement\ Interval}$$

Here the number of user-level retired instructions is a hardware event. A low-level metric based on hardware events is suitable for online feedback measurement. This kind of metric has three advantages. First, a low-level metric is application-agnostic and can be used to estimate the performance of various workloads simultaneously. Thereby, a resource manager does not need to instrument different applications or individually handle various application-level metrics. Second, a low-level metric can be measured conveniently without modifying any application or OS kernel. Most modern microprocessors provide PMUs that can be used to monitor hardware events. Operating systems like Linux already provide an interface to read the PMUs, and the extra measurement overhead is negligible [73]. Third, this metric is fine-grained and can provide meaningful information in a short time interval. With frequent performance measurements, a resource manager can provide timely and accurate reactions. More importantly, performance changes can be measured in the process of workload execution. Thereby, the UIPT metric is suitable for measuring the performance of an application that processes a long running workload, e.g., an OLAP database workload.

However, low-level metrics based on hardware events proposed so far are not well studied in the context of online performance measurement for adaptive resource allocation. Instructions per cycle (IPC) or its reciprocal cycles per instruction (CPI) is the most used low-level system performance metric. UIPT has a ratio structure similar to IPC, but UIPT is a better measurement of workload progress than IPC because of their differences in two important aspects.

First, UIPT is calculated using wall-clock time instead of CPU clock cycles, whereas IPC is the ratio of instructions to CPU cycles. An absolute timing metric is required to calculate the progress speed of a workload. However, CPU cycles do not represent the absolute length of time. A cycle is determined by the processor's hardware signals. CPU frequencies may (independently for individual cores) change while a program is executed. UIPT uses absolute timing independent of the changes in CPU frequency. Moreover, hardware performance counters only count cycles when the CPU is not in a halted state due to a STPCLK or HLT instruction. If an OS halts the CPU when the system is idle, the idle time is factored out of the IPC calculation, which thereby potentially overstates the progress speed of a workload. Otherwise, if an OS runs an idle loop instead of halting, the workload IPC must be influenced by the IPC of the idle loop. Finally, for a multi-core machine, it is not clear whether the total or average number of cycles across cores should be used in the calculation. The Linux perf tool uses the total number of cycles to calculate IPC when it measures multiple cores. For example, in the experiment with TPC-E database workload in Section 2.2.1, the number of cycles continues to increase with the number of cores. The number of retired instructions and application performance stops increasing when the core allocation exceeds a certain value. Thereby, with excessive core allocation, IPC continues to decrease regardless of how the application performance changes. In this case, IPC reflects only the efficiency of multi-core platforms, but is not meaningful as a progress metric. In contrast, UIPT is calculated using wall-clock time, hence the inaccuracy due to cycle measurement can thus be avoided.

Second, only user-level instructions are counted in UIPT, whereas kernel-level instructions are excluded. Not all the retired instructions should be counted when workload progress is accessed. Multithreaded applications, such as database servers, typically use OS-level threads to utilize hardware parallelism. With a highly concurrent workload, these threads might wait for synchronization primitives, and the resulting operations typically happen in the OS kernel. These operations do not process actual work in the application and thus the number of their instructions does not correlate with application performance. In UIPT, the number of user-level instructions is used to estimate the number of instructions that make progress in the workload. The result using user-level instructions is better than or at least equal to that using total instructions.

Only when a considerable number of instructions are retired in user-level synchronization routines (thus not productive), UIPT fails to accurately reflect the workload progress. However, under this circumstance, alternatives such as the total instructions and kernel-level instructions have the same problem and user-level instructions are still better. Moreover, UIPT can indicate the change in execution speed of an application using user-level synchronization. For instance, lock-free programming relies on atomic instructions rather than synchronization primitives in the kernel. When the parallelism of a lock-free program increases, the CPU cycles spent on atomic instructions increase (e.g., as shown in Figure 2.3). Therefore, the workload progress slows down and UIPT also decreases. The Canneal benchmark program from the Parsec suite [6] uses a lock-free algorithm. The evaluation result shows that there is a strong correlation between the changes in UIPT and the execution time of this program (see Table 3.1 in Section 3.4).

UIPT represents low-level workload throughput itself. Meanwhile, it can also reflect the progress speed of a workload, therefore UIPT can also be used to indirectly estimate workload execution time. While IPC is a general measure of computational efficiency, it is not meaningful as a progress metric on modern multi-core platforms. Previous studies in the literature report that the classical IPC metric may inaccurately reflect performance and lead to incorrect or misleading conclusions for multithreaded programs on multi-processor systems [1]. IPC or its reciprocal CPI may be used to detect the changes in application performance, but does not show a strong correlation with application-level metrics. Zhang et al. propose a tool based on CPI to detect the interference between tasks on Google's clusters [78]. However, they do not evaluate the correlation between CPI and other performance metrics. Instead, in their paper, they show that TPS of a batch job has a strong correlation with IPS (instructions per second). IPS and CPI should be considered as two different metrics because IPS uses wall-clock time. Actually the idea of IPS is similar to that of UIPT, but IPS is not studied further in their paper.

3.3 Relative Change

The feedback-based allocation algorithms proposed in this thesis do not rely on the absolute value of a metric to find the optimal CPU allocation. The best application performance depends on the specific workload. Typically, an allocation algorithm has no knowledge about the possible best application performance. This thesis assumes that every application has a concave performance curve with different CPU allocations. The proposed algorithms estimate the application's current position in the curve and search for the optimal CPU allocation based on performance changes caused by changes in CPU allocation.

Moreover, the algorithms need to take different actions based on to what extent the application performance changes. Relative change is employed to describe the extent. The relative change between two values x and $x_{reference}$ is defined as

$$RelativeChange(x, x_{reference}) = \frac{x - x_{reference}}{x_{reference}}$$

Here $x_{reference}$ is the metric value before the CPU allocation changes. For instance, when one more core is allocated to the application, the relative change of a metric is calculated as

$$\frac{Value\ with\ N+1\ cores\ -\ Value\ with\ N\ cores}{Value\ with\ N\ cores}$$

The algorithms need to estimate the changes in application performance based on the changes in UIPT or other measured metrics. There must be a strong correlation between the relative change in the measured metric and the relative change in application performance. More details about the proposed algorithms will be introduced in Chapter 4. This chapter mainly evaluates whether a metric is a good proxy for another metric based on the correlation between the relative changes in different metrics.

3.4 Experimental Evaluation

In this section, a group of experiments with various workloads are used to evaluate the metrics discussed in this chapter, and whether these metrics are suitable for online performance measurement is investigated. First, the limitations of CPU usage and utilization are verified. Next whether UIPT can be used to estimate different application-level metrics is tested.

The testbed machine introduced in Section 2.2 is used for the experiments in this section. With a total of 64 CPU cores, applications use up to 32 cores exclusively, while the other 32 cores are used for clients to generate requests. The cpuset subsystem of Linux cgroups is used to control which cores the applications can use. Applications use memory interleaving across NUMA nodes. Thus the memory interference is balanced for the workloads.

3.4.1 Evaluation of CPU Usage

This section evaluates whether CPU usage and CPU utilization can be used as a proxy for application-level performance metrics. Two different database workloads are used in the experiments. The database server and its system configuration used in this section is the same as that used in Section 2.2. The first workload is the transactional workload used in Section 2.2 and its performance metric is transactions per second. The database and transactions are from the TPC-E benchmark [12]. The second workload is a synthetic analytical workload based on queries from the TPC-H benchmark [13]. The application-level performance metric of this workload is the workload execution time. The database size is 1 GB[1]. TPC-H contains 22 queries of a business decision support system. These queries process large volumes of data, and each takes different time. The synthetic workload has 20 concurrent clients. Each client submits all 22 queries to the database server serially, but in a different random order. The concurrent client sessions do not complete at the same time. Therefore, at the end of each experiment run, some cores become idle while some are still busy. The average completion time of all the 20 client sessions is used to evaluate the correlation between metrics.

These workloads are repeated with the number of physical cores being varied from 1 to 32. CPU usage and the application performance with different numbers of cores are measured. CPU utilization is calculated from CPU usage divided by the number of allocated cores. For the TPC-E workload, each run in the experiment lasts for 240 seconds to collect measurement samples. TPS and CPU usage are measured every 10 seconds. For the TPC-H workload, execution time can only be measured when the workload is finished. Therefore this workload is repeated 20 times for each CPU allocation to observe the variation. With a static core allocation, the measurements of CPU usage and application performance are stable. The results shown in the following figures are the mean of 20 samples. The coefficient of variation (CV) of all results (application performance, CPU usage and CPU allocation) are less than 0.06.

The Pearson product-moment correlation coefficient [61] is used to measure the association between different metrics in this section. It has a value between $+1$ and -1 where $+1$ is in the case of a perfect direct linear correlation, while -1 is a perfect inverse linear correlation. Besides the correlation coefficient between the values of metrics, this study also calculates the correlation coefficient between the relative changes in different metrics

[1] The TPC-H workload is also used in the experiments with multiple workloads in Chapter 5, in which 1GB database size provides a proper length of experiment time. The experiment in this section uses this size for a consistent presentation. With a larger database size, UIPT also has a good correlation with the execution time of this workload [28]

when the number of cores is increased from 1 to 32.

Figure 3.1 presents the average CPU usage and average application performance for the TPC-E and TPC-H workloads. The CPU usage result is presented using the number of equivalent effective cores. For the TPC-E workload, the correlation coefficient between TPS and CPU usage is 0.78. CPU usage keeps increasing regardless of the change in TPS. When the CPU allocation is more than the optimal point, extra CPU resources are not directly used to process the work from clients.

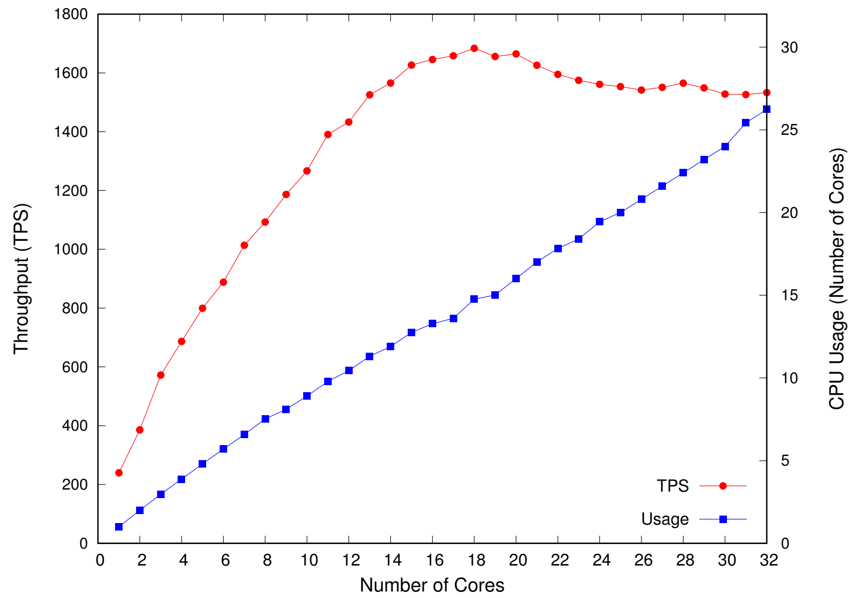
When the number of cores is increased from 1 to 32, the relative change in CPU usage shows a good correlation with the relative change in TPS. The coefficient is 0.96. However, the good correlation between relative changes stems from the results with 1 to 16 cores. Considering only the results with 15 to 32 cores, the correlation coefficient is 0.38. As a consequence, it is difficult to find the optimal CPU allocation (18 cores for this workload) in the range of 15 to 32 cores based on the changes in CPU usage.

The TPC-H workload employs only 20 clients and uses at most 20 effective cores (see Figure 3.1(b)). The average execution time of 20 client sessions is used to calculate the correlation coefficient, but the minimum and maximum execution time are also depicted in the figure. The correlation coefficient between execution time and CPU usage is -0.40. The correlation is weak. The shortest execution time is not achieved when the CPU usage reaches the peak.

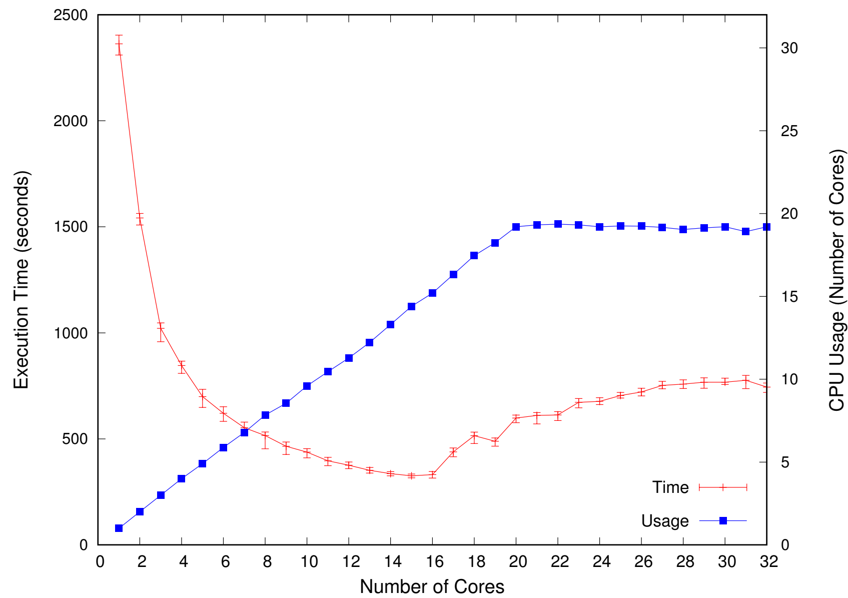
The correlation coefficient between the relative changes in the two metrics is -0.71 when the cores are increased from 1 to 32. With more than 20 cores, the execution time continues to change when the CPU usage does not change. Similar to the TPC-E case, the change in CPU usage cannot provide an accurate estimation of performance change in the experiment with the TPC-H workload. CPU usage is not a good proxy for the two application-level metrics.

The relationship between CPU utilization and application-level performance metrics is studied in the same way using the two database workloads. For the TPC-E workload, the results of average TPS and average CPU utilization are shown in Figure 3.2(a). The correlation coefficient between the average TPS and the average CPU utilization is -0.94, which shows an inverse linear correlation.

However, the correlation coefficient between relative changes in the two metrics is only -0.09. There is almost no correlation between the relative changes. From the curves in Figure 3.2(a), it is apparent that when the TPS results change dramatically, CPU utilization sometimes does not change correspondingly, e.g., when the number of cores is changed from 8 to 10. Hence it is difficult to estimate the changes in TPS based on the

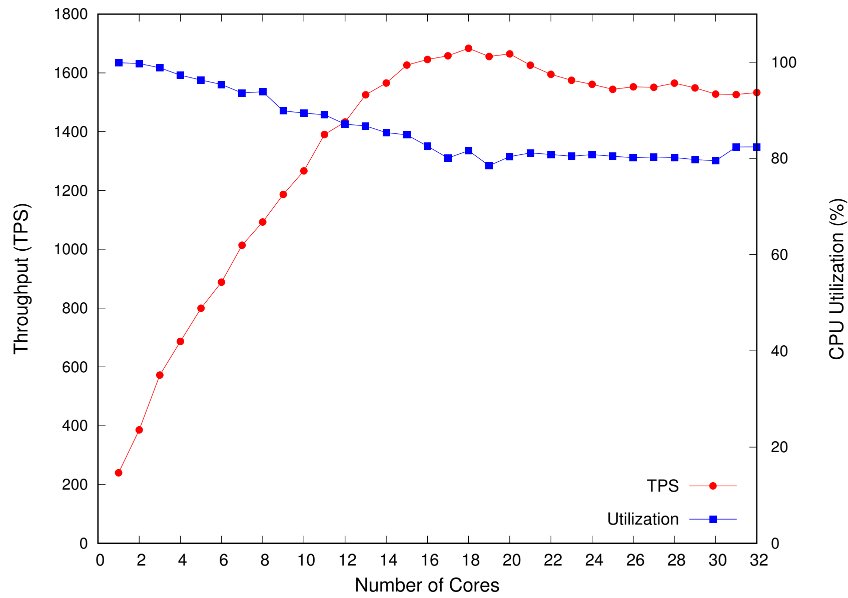


(a) CPU Usage for TPC-E Workload

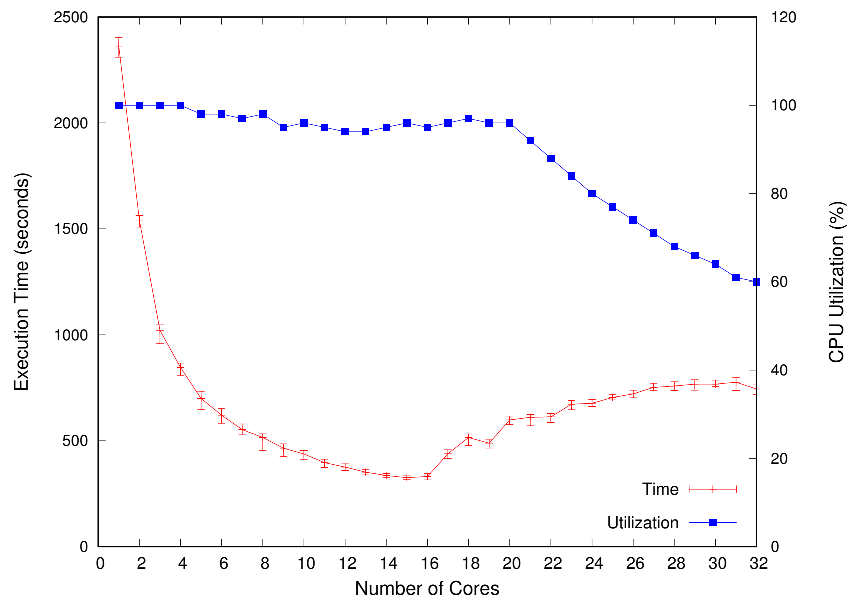


(b) CPU Usage for TPC-H Workload

Figure 3.1: CPU Usage for Database Workloads



(a) CPU Utilization for TPC-E Workload



(b) CPU Utilization for TPC-H Workload

Figure 3.2: CPU Utilization for Database Workloads

changes in CPU utilization. CPU utilization cannot provide a solid foundation to decide whether to increase the CPU allocation to the TPC-E workload.

For the TPC-H workload, what can be clearly seen in Figure 3.2(b) is the weak correlation between the average execution time and the CPU utilization. The correlation coefficient between the two metrics is only -0.01. The correlation coefficient between relative changes in the two metrics is -0.14. The relationship between the relative changes is irregular. For example, with 1 - 8 cores, the CPU utilization is high (close to 100%), while the execution time is very different. Hence it is hard to estimate how the execution time changes based on the change in CPU utilization. In summary, CPU utilization is not a good proxy for the two application-level metrics either.

3.4.2 Evaluation of UIPT

The relationship between UIPT and application-level performance metrics is evaluated using the same two database workloads. In the experiments, UIPT is measured every second while the database server is processing the workloads. The coefficient of variation of UIPT results with a fixed number of cores is at most 0.07.

The results of average TPS and average UIPT with 1 to 32 cores are illustrated in Figure 3.3. It can be seen that the two metrics show the same trend. The correlation coefficient between UIPT and TPS is 0.99. More importantly, The turning of the two curves occur simultaneously. The correlation coefficient between the relative changes in UIPT and the relative changes in TPS is also 0.99. As a comparison to CPU usage, if considering only the results with 15 to 32 cores, the correlation coefficient between the relative changes in TPS and UIPT is 0.87. Such a strong correlation makes UIPT a good proxy of TPS for this kind of transactional workload.

For the TPC-H workload, the execution time and UIPT with different numbers of cores are presented in Figure 3.4. The two curves clearly show an inverse correlation. When the UIPT value is higher, the average execution time is shorter. The correlation coefficient between average execution time and UIPT is -0.84. More vital is the correlation between the relative changes in the two metrics. The coefficient is -0.94, which shows a strong inverse linear correlation. Thus, UIPT is also a good metric to estimate the performance changes of this analytical database workload.

When the TPC-E and TPC-H workloads achieve their best performance, they do not have the same absolute value of UIPT. However, a resource manager can search for the best CPU allocation based on the relative changes in UIPT regardless of the specific workload,

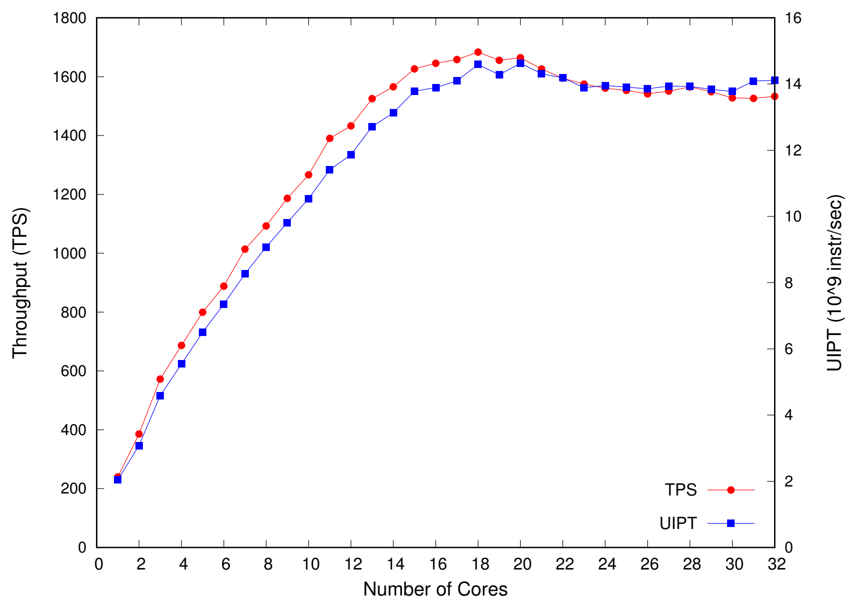


Figure 3.3: UIPT for TPC-E Workload

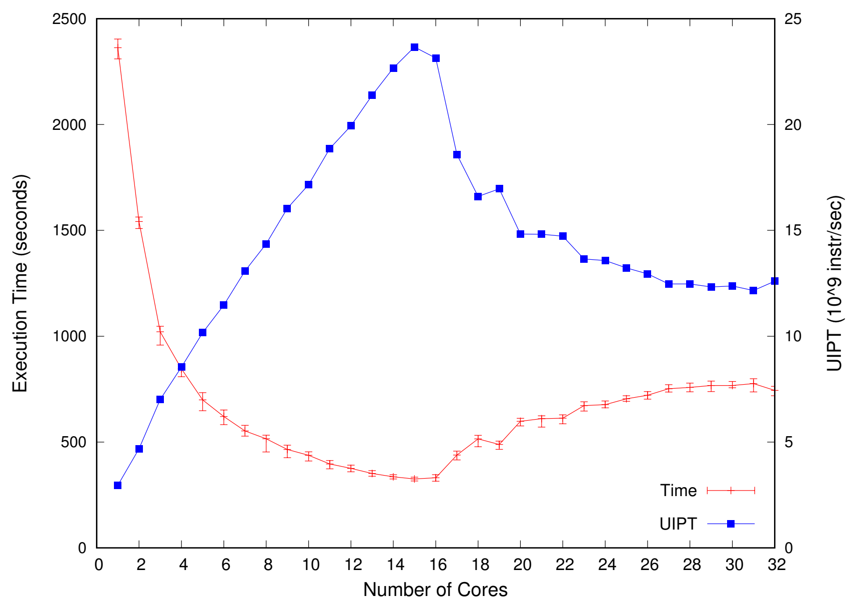


Figure 3.4: UIPT for TPC-H Workload

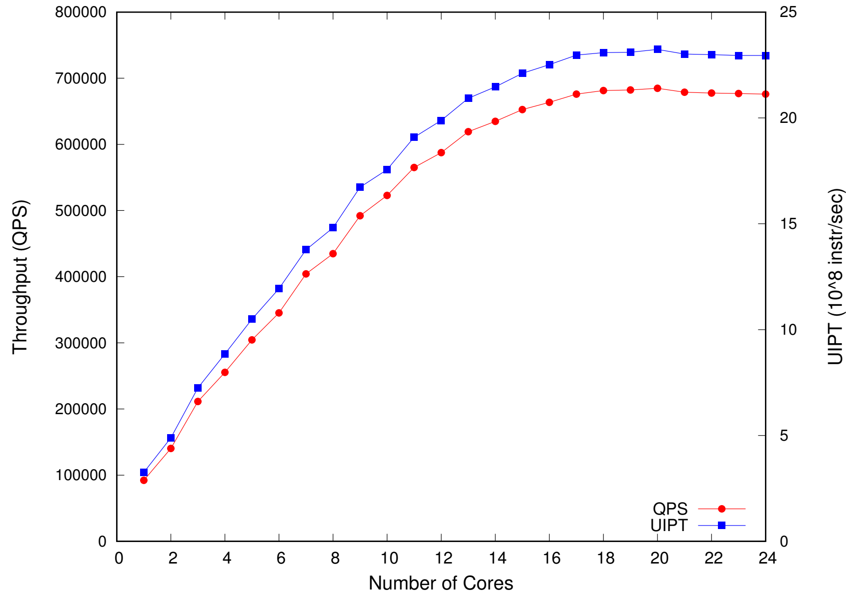


Figure 3.5: UIPT for Memcached Workload

because there exists a strong correlation between relative changes in UIPT and relative changes in application-level metrics.

UIPT also performs well for other software systems. In Section 2.2.1, a write-heavy Memcached workload is employed to study the impact of excessive parallelism. In this section, UIPT and the system throughput QPS are measured with the same workload. The core allocation is varied from 1 to 24 via the cpuset subsystem of Linux cgroups. This workload is repeated 20 times with each core allocation. Only the average results are shown in Figure 3.5, since the coefficient of variation is less than 0.01 for QPS, and less than 0.04 for UIPT. UIPT has a strong correlation with QPS for this Memcached workload, and the coefficient is greater than 0.99. The correlation coefficient between the relative changes in the two metrics also reaches 0.99.

The evaluation is then expanded to other parallel programs. The PARSEC benchmark suite consists of 13 programs from a wide range of areas, such as computer vision, video encoding, financial analytics, animation physics and image processing [6]. This group of experiments use a fixed number of threads while varying numbers of cores. All programs are executed with 256 threads, except that blackscholes runs with 128 threads[2]. The native input data set for execution on real machines is used in the experiments. The number of

[2]When the number of threads is 256, the blackscholes program crashes on the testbed machine.

Table 3.1: Parsec Correlation Results

Program	Correlation between Time and UIPT	Correlation between Relative Changes
blackscholes	-0.86	-0.99
bodytrack	-0.95	-0.99
canneal	-0.97	-0.99
dedup	-0.99	-0.99
facesim	-0.99	-0.99
ferret	-0.80	-0.99
fluidanimate	-0.89	-0.99
freqmine	-0.84	-0.99
raytrace	-0.96	-0.99
streamcluster	-0.96	-0.98
swaptions	-0.99	-0.99
vips	-0.86	-0.99
x264	-0.89	-0.98

cores is varied from 1 to 32. The correlation results are shown in Table 3.1. In most cases the negative correlation between execution time and UIPT is very clear. More importantly, the strong correlation between the relative changes in the two metrics means that changes in UIPT can be used to estimate the performance changes of these typical parallel programs.

In cloud computing, applications may run in a virtual machine or a container. UIPT of applications can be measured outside the virtualisation layer. The evaluation is thus expanded to virtualisation environments. First, a virtual machine using KVM [42] and libvirt [29] is tested. An original MariaDB server is launched in the virtual machine. CPU allocation is controlled by the cpuset subsystem of Linux cgroups. The same TPC-E workload is measured with a varying number of cores. Figure 3.6 shows the average TPS and average UIPT results with 1 to 32 cores. The correlation coefficient between UIPT and TPS is higher than 0.99, and the correlation coefficient between the relative changes in the two metrics is also higher than 0.99. Second, the same experiment is conducted using a Docker container [19]. The docker image of MariaDB (tag 10.0) from Docker Hub is used [20]. UIPT still shows a strong correlation (coefficient > 0.99) with the throughput of this database workload. The correlation coefficient between the relative changes in the two metrics is higher than 0.99. The measurements of the two metrics with different core allocations are shown in Figure 3.7. The results of the two experiments confirm

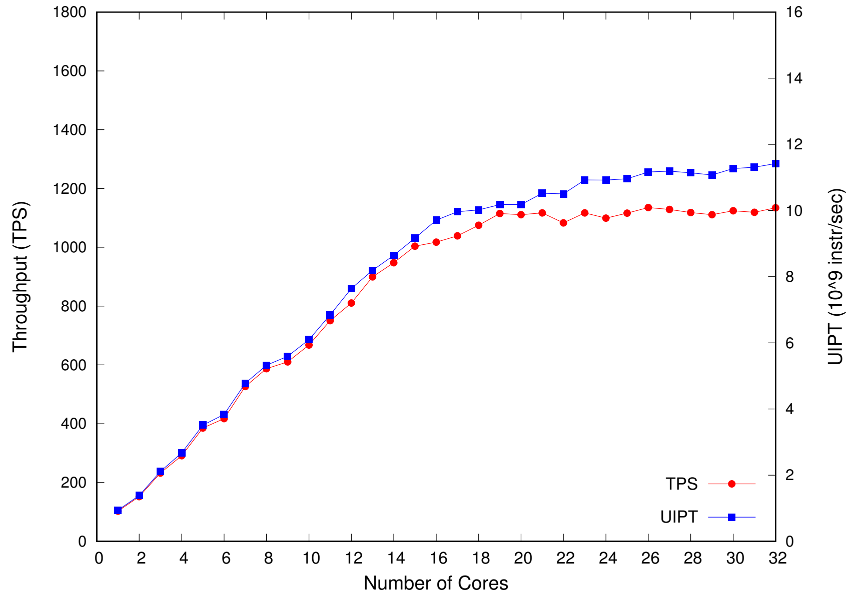


Figure 3.6: UIPT for TPC-E Workload in a VM

that UIPT can be used to estimate the performance of applications in these virtualisation environments.

The measurement of UIPT is independent of the way how CPU resources are allocated. The Linux perf_event interface [73] can track the user-level instructions retired by threads in a cgroup. Therefore, even when CPU resources are allocated to multiple tenants with the quota mechanism in Linux cgroups, UIPT can still be used to estimate the changes in application performance. For example, in the experiments with the TPC-E database workload mentioned above, if the CPU allocation is adjusted via the quota mechanism, the correlation between UIPT and application-level metrics is still higher than 0.99. Figure 3.8 illustrates that the average TPS and the average UIPT have the same trend with varying quota values. The correlation coefficient between the relative changes in the two metrics is higher than 0.99.

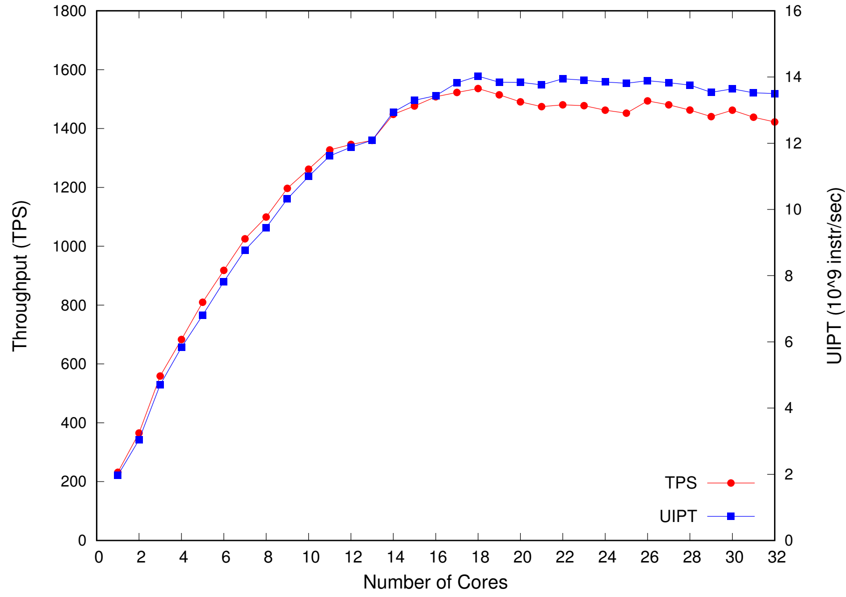


Figure 3.7: UIPT for TPC-E Workload in a Docker container

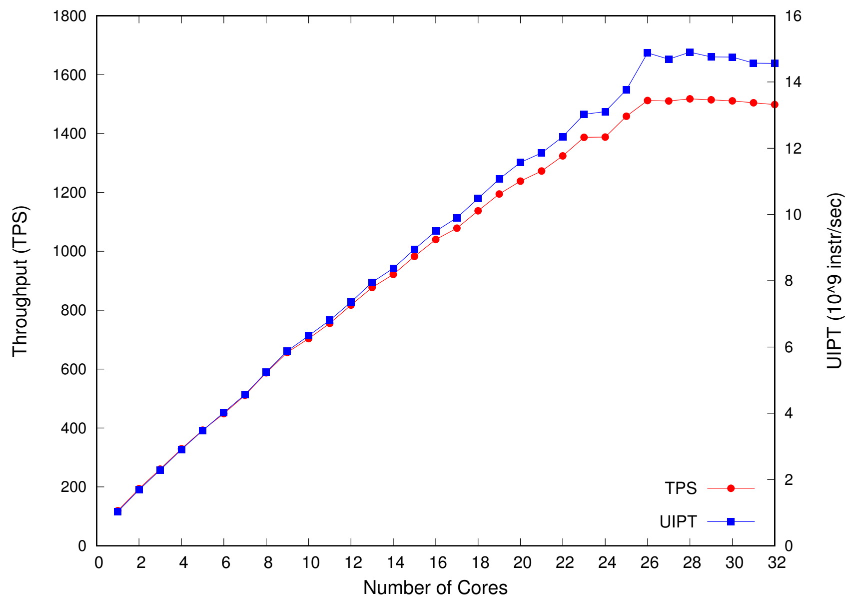


Figure 3.8: UIPT for TPC-E Workload with Different Quota Values

Chapter 4

Adaptive CPU Allocation

This chapter studies adaptive CPU resource allocation based on the UIPT metric as a feedback signal. Adaptive CPU allocation enables a resource manager to simultaneously achieve resource isolation and work conservation. On the one hand, the resource manager enforces CPU resource isolation by restricting the maximum amount of CPU resources available to each workload. On the other hand, the restriction is relaxed according to requirements of other workloads to make full use of otherwise idle CPU resources.

Adaptive CPU allocation requires an allocation algorithm to detect the CPU requirement of a workload automatically. Real-world workloads are highly dynamic, so it is impractical for users to determine the resource requirements manually. The adaptive resource allocation problem is modelled as a dynamic optimization problem in this thesis. Two algorithms are proposed in this chapter to track the movement of the optimal allocation. Both of these algorithms allocate CPU resources based on how the application performance changes with different CPU allocations. The application performance is measured via UIPT, so the algorithms can support various applications and make timely adjustments.

4.1 Problem Definition

4.1.1 Performance Curve of A Stable Workload

There is no perfectly scalable software system. Allocating more CPU resources to an application does not necessarily result in better performance. Beyond a certain point, the marginal change in application performance becomes relatively small or even negative.

There are two possible reasons for this phenomenon. First, the software system may be unsaturated. No more work needs to be done in parallel, and thus the system does not consume more CPU resources. Second, it is also possible that certain bottlenecks in the system prevent many threads from running in parallel. Thereby the system is not scalable any more when CPU allocation is more than a certain amount. If an application has achieved its target performance, there is no need to allocate more CPU resources to this application. In a multi-tenancy scenario, multiple tenant workloads are consolidated on the same infrastructure. Idle CPU resources can be utilized by other workloads, thereby increasing the output of the whole infrastructure. Even if there is no other workload demanding more CPU resources, the frequency of idle CPU cores can be scaled down to reduce power consumption.

However, certain parallel software systems, e.g., database systems, are highly complex and dynamic, making it challenging to allocate appropriate hardware resources for these systems to achieve optimal performance. The manual tuning process is usually time consuming and skills intensive for system administrators. More importantly, a static configuration hardly meets the time-varying requirement of a workload. Therefore, the problem addressed in this chapter is how to automatically and dynamically allocate CPU resources to a workload according to its requirement.

For a stable workload, as the CPU allocation increases, the system performance initially improves, but the trend gradually stops or even reverses. This trend is verified by the experimental results in previous sections. For instance, Figure 2.1 and Figure 2.4(b) in Section 2.2.1 show how the throughput of two different systems changes with the number of cores, and Figure 3.4 in Section 3.4 shows how the execution time of an analytical workload changes with varying cores. In these experiments, every software system uses a static system configuration. Every workload has a stable request arrival rate and static characteristics, e.g., the same queries. The number of cores is assumed to be the only independent variable, and thus the system performance can be modelled as a function of CPU allocation for a stable workload.

The performance function $f(x)$ of the number of cores x can be approximated by a concave-downward curve whose derivative $f'(x)$ monotonically decreases with increment in the number of cores. The allocation problem for a stable workload is to find the stationary point of f, which is the solution to $f'(x) = 0$, also known as the root of the derivative. As the derivative monotonically decreases, a simple local search strategy can be used to find the stationary point. The search strategy is to increase x when $f'(x) > 0$ while decreasing x when $f'(x) < 0$. Instead of modelling the unknown $f(x)$ and solving the equation $f'(x) = 0$, the approach in this proposal uses $\frac{\Delta f}{\Delta x}$ to estimate whether $f'(x)$ is positive or negative. In the search, if adding cores by Δx causes Δf to become negative,

the number of cores must be decreased by Δx; otherwise more cores ought to be added. The search from the opposite direction is similar. When the number of cores is discrete, the exact stationary point may not be reached. The searching procedure will stop at the integer point closest to the stationary point. The search step (Δx) can adapt to the value of the derivative, i.e., Δx can be changed to a larger value when $|\Delta f|$ is large, because a large $|\Delta f|$ implies that the system is far from the stationary point.

The performance curve of a real-world complex system is not a perfectly smooth concave curve like a parabola. For example, the throughput of a database system may not decline dramatically when the number of cores is larger than the stationary point, but fluctuate with a downward trend. This thesis assumes that the performance curve is essentially concave, but also takes possible anomalies into consideration. The two most common anomaly states are plateau and local optimum.

In the plateau state, all the neighbouring points have similar performance values when the number of CPU cores is larger than the stationary point. Therefore, it is unclear how to choose the next step by making local comparisons. This state means more CPU resources do not result in better system performance, because the system is unsaturated or not further scalable with respect to CPU resources.

Local optimum refers to the state in which a point has a better performance than all neighbours, but this point is not the point having the best performance. For example, in certain cases, the throughput of a database server does not increase apparently with one more core, but if more cores continue to be allocated, the throughput will eventually increase to a higher level.

4.1.2 Problem Model for Dynamics

Application workloads are highly dynamic in the real world [63]. At each moment the relationship between application performance and CPU allocations can still be modelled as a concave curve. However, when the temporal dimension is taken into account, some elements of the underlying model change in the process of searching for the optimal CPU allocation. In this thesis, finding the optimal CPU allocation for a dynamic workload is modelled as a dynamic optimization problem (DOP). A DOP is an optimization problem in which the objective function or the restrictions change over time. The general definition of a DOP is shown in Equation 4.1 [14].

$$DOP = \begin{cases} optimize f(x,t) \\ s.t.\ x \in F(t) \subseteq S,\ t \in T \end{cases} \tag{4.1}$$

where:

- $S \in R^n$, S is the search space
- t is time
- $f : S \times T \longrightarrow R$, f is the objective function that assigns a numerical value to each possible solution x ($x \in S$) at time t.
- F(t) is the set of feasible solutions, $x \in F(t) \subseteq S$ at time t

In the adaptive CPU allocation problem, S is the available CPU resources; f is the performance function of CPU allocations, and f will change over time; F is the feasible set, for example, a CPU allocation x can be only an integer when core partitioning is used, but F does not change over time in this adaptive allocation problem.

The goal of solving a DOP is no longer to find a stationary optimal solution but to have a mechanism to efficiently keep track of the movement of the function f. An evolutionary algorithm is the most common approach to this kind of dynamic optimization problem because it does not make any assumption about the underlying problem landscape. An evolutionary algorithm needs a fitness function to determine the quality of different solutions. In a real-world application, the disadvantage of an evolutionary algorithm is the cost of its fitness function. For the adaptive CPU allocation problem, an evolutionary algorithm determines the effect of a CPU allocation by applying this allocation. However, if an evolutionary algorithm changes CPU allocation randomly and drastically, the whole system becomes unstable and the application performance is unpredictable. Therefore, a gradual local search is preferred in this work to solve the adaptive CPU allocation problem, meaning that only a small modification is introduced every time.

The two allocation algorithms proposed in this chapter consider each workload change as the arrival of a new optimization problem and re-optimize. This study assumes there is no direct information about workload changes. The two algorithms keep searching along the performance curve based on performance changes and never enter a stable state. This search strategy is effective for the adaptive CPU allocation problem. The details about how the algorithms handle workload changes will be introduced in the following sections.

4.1.3 Regulation Problem

The allocation problem can also be modelled as a regulation problem. As introduced above, the allocation problem for a stable workload is to find the point where the derivative is

zero. The classical proportional-integral-derivative (PID) controller is a typical method to regulate the derivative to zero. However, a PID controller requires extensive tuning based on its target system. A real application or the application's performance curve is needed for the tuning [30]. Hence the resulting parameters are specific to an application or workload. These controller parameters impact rise time, overshoot and settling time. Preliminary investigations for this thesis have indicated that different applications require substantially different parameters. This study looks for a robust uniform mechanism that can handle different application simultaneously, and hence investigates simpler local search algorithms.

4.2 Hill Climbing

As shown in Figure 4.1, the whole allocation algorithm consists of two parts, an inner controller and an outer corrector. The inner controller is designed to search for the optimal allocation for a stable workload. The inner controller is simple and stable, while the outer corrector overwrites the output of the inner controller in response to workload changes. The whole algorithm has two inputs: the change in CPU allocation ($du(k)$) and the change in measured output ($dy(k)$) between the current and previous measurement intervals. The output of the algorithm is the CPU allocation to the target system in the next interval ($du(k+1)$). The output of the target system is measured in each interval and used to calculate $dy(k)$. In this work, the measured output is the low-level system throughput UIPT because this metric shows a strong correlation with application-level performance metrics (cf. Section 3.4).

This section proposes a hill-climbing algorithm as the inner controller. Hill climbing attempts to maximize (or minimize) a target function by making comparisons between results from different solutions. A straightforward hill-climbing algorithm starts with an arbitrary solution to a problem, and then attempts to find a better solution by incrementally changing a single element of the solution. If the change produces a better solution, an incremental change is made to the new solution, and this process repeats until no further improvements can be achieved.

In this thesis, the hill-climbing algorithm searches for the optimal CPU allocation for a stable workload. The design is based on the assumption that the performance function $f(x)$ of CPU allocation x has a concave curve. This algorithm implements the search strategy in the previous section, which changes x according to the value of $\frac{\Delta f}{\Delta x}$. The algorithm is based on the relative change rather than the absolute change Δf so that the parameters in the

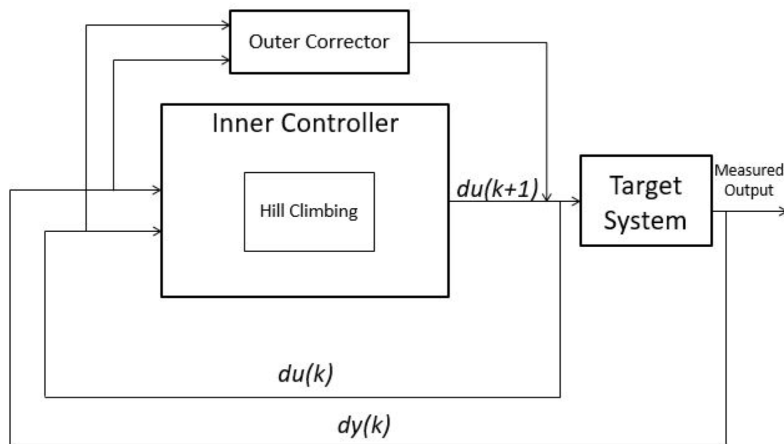


Figure 4.1: Architecture of the Allocation System

algorithm are independent of the absolute measurements. The CPU allocation is repeatedly adjusted based on the relative change in UIPT compared to the last measurement.

In the algorithm descriptions in this and the next section, CPU allocation is changed by the number of cores. With core partitioning, the allocation change must be an integer number of physical cores. With quota-based CPU allocation, the change is a number of effective cores, which is calculated by the quota value over the allocation period. The number of effective cores can be an integer or a fraction. Theoretically, the algorithm can change CPU allocation by a fraction of effective core.

The basic hill-climbing algorithm is shown in Algorithm 1. It uses a threshold th_{search} to determine whether the previous relative change in UIPT is significant enough to request a change in CPU allocation during the next round. The th_{search} is set to 0.04 based on the results of preliminary experiments. This value performs well in the evaluation experiments with different workloads.

A simple hill-climbing algorithm does not guarantee to find the global optimum. If the curve is not perfectly concave, such an algorithm may converge to an inappropriate state. In practice, even a stable workload does not necessarily have a perfect performance curve, and a workload change makes the curve change. The hill-climbing algorithm is driven by changes in application performance that occur in response to changes in CPU allocation. However, changes in application performance may also be caused by workload changes.

How a feedback-based allocation algorithm handles a workload change depends on the

Algorithm 1 HillClimbing(dy, du)

dy: change in performance (UIPT)
du: change in cores in the previous round
th_{search}: change threshold

1: **if** $dy > th_{search}$ **then**
2: $dC \leftarrow \text{sign}(du) * 1$
3: **else if** $dy < -th_{search}$ **then**
4: $dC \leftarrow \text{sign}(du) * -1$
5: **else**
6: $dC \leftarrow 0$
7: **end if**
8: return dC

magnitude and frequency of the measured performance change arising from the workload change. When the measured performance change is very small, the workload change is negligible meaning that no algorithm response is required. When the measured performance change is very large, the performance change should be caused by a workload change rather than by a change in CPU allocation. When neither of the above two cases applies, the magnitude of the measured performance change caused by the workload change is close to that caused by the change in CPU allocation. In this case, a hill-climbing algorithm (or any other feedback-based algorithm) cannot tell whether the measured performance change is caused by the change in CPU allocation. If this kind of workload changes happen at the frequency of measurement, and the spurious performance changes are continuously measured, a feedback-based algorithm will be disrupted and incapable of performing properly. To deal with these curve anomalies and workload changes, the outer corrector incorporates several fixes. The complete allocation algorithm is shown in Algorithm 2.

The algorithm uses a change threshold th_{search} to determine whether the performance change is small enough and uses a demand change detection threshold th_{demand} to determine whether the performance change is large enough. When the measured change in UIPT is higher than the th_{search} threshold but does not exceed the th_{demand} threshold, the hill-climbing algorithm cannot distinguish whether the UIPT change is caused by a change in workload or a change in CPU allocation. Meanwhile, if the frequency of these workload changes is close to the frequency of feedback measurements, spurious UIPT changes will be measured successively. The highly frequent UIPT changes continuously keep the hill-climbing algorithm searching in a sub-optimal range without converging to the optimal

Algorithm 2 Allocate(du)

du: change in cores in the previous round
P_t: Performance (UIPT) measured in interval t
C_t: core allocation in interval t
w: number of intervals of multi-interval hill climbing
th_{greedy}: greedy core reduction threshold
th_{demand}: demand change detection threshold

1: **if** $(t \mod w) = 0$ **then**
2:     /* averages $\overline{U}$ and $\overline{C}$ computed over last two w periods */
3:     $dy \leftarrow (\overline{P}_t - \overline{P}_{t-w})/(\overline{P}_{t-w})$
4:     $dC \leftarrow HillClimbing(dy, \overline{C}_t - \overline{C}_{t-w})$
5: **else**
6:     **if** $C_t = C_{t-1}$ **then**
7:         $dC \leftarrow random$ from $\{-1, 0, 1\}$
8:     **else**
9:         $dy \leftarrow (P_t - P_{t-1})/(P_{t-1})$
10:         $dC \leftarrow HillClimbing(dy, C_t - C_{t-1})$
11:         **if** $dC = 0$ **and** $C_t < C_{t-1}$ **and** $dy > th_{greedy}$ **then**
12:             $dC \leftarrow -1$
13:         **end if**
14:     **end if**
15: **end if**
16: **if** $du > th_{demand}$ **then**
17:     $dC \leftarrow 2$
18: **else if** $du < -th_{demand}$ **then**
19:     $dC \leftarrow -C_t/2$
20: **end if**
21: return dC

allocation. Typical techniques that randomly restart hill climbing in another range bring instability to CPU allocation and make application performance unpredictable. This work proposes a multi-timescale design to handle high-frequency workload changes. A basic hill-climbing algorithm allocates cores according to the UIPT measured every interval. A second hill-climbing instance operates on the average UIPT and average core allocation over multiple measurement intervals to make adjustments at a larger timescale (Lines 1-4 in Algorithm 2). The second hill-climbing instance at a larger timescale is named multi-interval corrector in this design. The adjustments from the multi-interval corrector happen less frequently, but take precedence over the inner basic algorithm. This multi-timescale mechanism also keeps the inner algorithm constantly searching, which makes the whole algorithm adaptive to workload changes. The two timescales do not need to be far apart. In the evaluation experiments in Section 4.4, a combination of 1- and 3-second timescales already achieves satisfactory results.

In contrast, if the interval between successive workload changes is much longer than the feedback measurement interval, the basic agility of the feedback loop will typically handle the change and adjust the CPU allocation accordingly. To further support this agility, the allocation algorithm never enters a stable state for a long period of time. If the CPU allocation does not change in one step ($C_t = C_{t-1}$), the hill climbing does not work, and then the next step is chosen randomly (Lines 6-7 in Algorithm 2).

Moreover, an additive increase multiplicative decrease (AIMD) strategy is used to deal with significant workload changes (Lines 16-20 in Algorithm 2). Specifically, if the absolute value of the relative change in UIPT exceeds the th_{demand} threshold (typically chosen as 0.5), this change is considered to be caused by a workload change rather than the recent adjustment of CPU allocation. When the relative change is positive, two more cores will be allocated to this workload. Otherwise half of the current allocation will be reduced. The AIMD mechanism makes it possible for the algorithm to react quickly to a significant workload change.

In addition, certain applications exhibit a plateau in their performance curve, where their performance does not change significantly when additional cores are allocated. To improve the system efficiency, i.e., to use as few CPU resources as possible, the algorithm tries to find the knee point of the plateau. After the removal of one core, if the change in UIPT is between $\pm th_{search}$, the basic hill-climbing algorithm does not take any action. Meanwhile if the change is greater than a lower bound th_{greedy} (typically -0.01), the previous removal does not result in a considerable performance degradation, so the algorithm continues removing another core (Lines 11-13 in Algorithm 2).

4.3 Fuzzy Control

This section introduces an alternative feedback-based algorithm to search for the best CPU allocation for a stable workload. In particular, this work uses an optimization algorithm based on fuzzy control, because the inherent nonlinearity of a computing system makes the design of a classical model-based feedback controller very challenging [30]. Such a classical controller is typically used for regulatory control, that is, to keep the performance metric at a desired reference value.

In contrast, the advantage of fuzzy control is that an accurate model of the target system is not required. Fuzzy control uses qualitative variables to describe the target system and controller actions, so it is easier to incorporate human expert knowledge on resource management in the form of fuzzy rules. Fuzzy control handles the uncertainty caused by measurement noise and system disturbances in a natural way via multiple fuzzy sets. The control objective for fuzzy control can be optimization instead of regulation [30]. Since a reference input is not necessary in fuzzy control, fuzzy control is suitable for an optimization problem.

The overall architecture of the fuzzy controller is the same as the architecture of the hill-climbing algorithm in Section 4.2. Specifically, the inner controller uses the fuzzy-control algorithm proposed in this section to replace the hill-climbing algorithm, while the outer corrector does not change (see Figure 4.2). The feedback loop operates in discrete time. The fuzzy controller has the same two inputs: the change in control input ($du(k)$) and the change in measured output ($dy(k)$) between the current and previous intervals.

The inner controller has three important steps as shown in Figure 4.2. Fuzzy rules are central to fuzzy control. Fuzzy rules are expressed using linguistic variables in the "IF-THEN" form. These qualitative rules do not contain numeric values. Therefore a fuzzification process first converts numeric inputs into fuzzy sets that quantify the rules. Fuzzy sets are associated with membership functions that quantify the certainty that input values may be classified as linguistic values. Inference in the second step determines which rules are relevant and to what extent, and draws a conclusion using the rules. Finally, a defuzzification method aggregates the outcomes of all implied fuzzy rules and inversely translates the resulting fuzzy set to a single numeric value.

The algorithm details are illustrated in Algorithm 3. When the control system initializes, the inner controller based on fuzzy control starts searching for the optimal number of cores for the target system. The outer corrector handles the plateau area and various workload changes. The multi-interval adjustment in the corrector prevents the algorithm from stopping at any stable state.

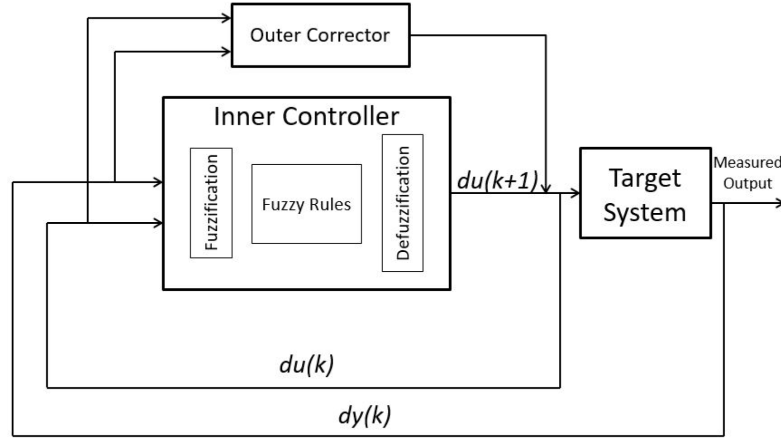


Figure 4.2: Architecture of the Controller

Algorithm 4 presents the basic components of the inner fuzzy controller. A *Fuzzification* function translates numeric measurements into inference results with certainty values. The inferences are made based on predefined fuzzy rules. Last, an AVG function combines the inference results to get a conclusion. The design details are explained as follows.

First of all, three linguistic variables are used to describe two discrete inputs and one output of the inner controller. The variable *change-in-cores* describes how the number of cores changes in the current interval (refer to $du(k)$ in Figure 4.2), and it takes on the two values: decrease and increase, in spite of how many cores decrease or increase. Another variable *change-in-throughput*, as the name implies, is the relative change in UIPT between the current and previous intervals ($dy(k)$). To describe to what extent the throughput changes, this variable employs four values: decrease, decrease_large, increase, and increase_large. The third variable *next-change-in-cores* is the change in CPU allocation to the target system, i.e., $du(k+1)$. Like *change-in-throughput*, it also has four values. The actual change in cores may be zero as a result of multiple rules.

Table 4.1 shows the "IF-THEN" form fuzzy rules applied in the controller. The first four fuzzy rules are basic rules that search for the optimal point in a curve. They cover the four situations described in the search procedure in Section 4.1. The last four rules just enlarge the search step Δ when *change-in-throughput* is large and the system is far from the optimal point.

Membership functions are used to quantify all three linguistic variables. As shown in Figure 4.3, triangular membership functions are used in this design. To get rid of the im-

Algorithm 3 Allocate(du)

P_t: performance (UIPT) measured in interval t
C_t: core allocation in interval t
du: previous change in cores
dy: relative change in UIPT
dC: change in cores
th_{greedy}: greedy core reduction threshold
th_{demand}: demand change detection threshold

```
1: if (t mod w) = 0 then
2:     dy ← (P̄_t − P̄_{t−w})/(P̄_{t−w})
3:     dC ← HillClimbing(dy, C̄_t − C̄_{t−w})
4: else
5:     dy ← (P_t − P_{t−1})/(P_{t−1})
6:     dC ← Fuzzy(dy, du)
7:     if du < 0 and dC = 0 and dy > th_greedy then
8:         dC ← −1
9:     end if
10:    if dy > th_demand then
11:        dC ← 2
12:    else if dy < −th_demand then
13:        dC ← −C_t/2
14:    end if
15: end if
16: return dC
```

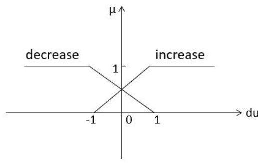
Algorithm 4 Fuzzy(dy, du)

FS: consequents of control rules
PS: certainty of consequents
gdu: gain factor for du
gdy: gain factor for dy
gdC: gain factor for dC

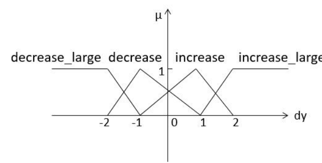
```
1: Fuzzification(du*gdu, dy*gdy, FS, PS)
2: dC ← AVG(FS, PS)
3: return Round(dC*gdC)
```

Table 4.1: Fuzzy Rules

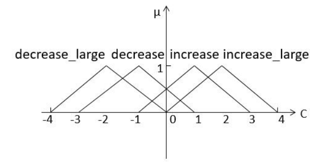
Rule	IF		THEN
	change-in-cores (du)	change-in-UIPT (dy)	next-change-in-cores (dC)
1	increase	increase	increase
2	increase	decrease	decrease
3	decrease	decrease	increase
4	decrease	increase	decrease
5	increase	increase_large	increase_large
6	increase	decrease_large	decrease_large
7	decrease	decrease_large	increase_large
8	decrease	increase_large	decrease_large



(a) change-in-cores



(b) change-in-throughput



(c) next-change-in-cores

Figure 4.3: Membership Functions

pact of measurement noise and system disturbances, uncertainty is introduced to describe to what extent the change in throughput is caused by the change in cores. The membership functions characterize this certainty/uncertainty. For example, in Figure 4.3(a), when the actual *change-in-cores* is 0, it is considered as both "increase" and "decrease". In Figure 4.3(b), if $dy = 0.5$ then $\mu_{increase}(0.5) = 0.75$ and $\mu_{decrease}(0.5) = 0.25$, so the controller is more certain that the throughput increases. After getting the certainty of each premise term in a rule via the membership functions, the minimum function is used to calculate the certainty of the conclusion of a rule, i.e., $\mu_{conclusion} = min(\mu_{change-in-cores}, \mu_{change-in-UIPT})$.

The input values to the membership functions are not the measurements from the system. All the input measurements are normalized before the membership functions are applied. The measured values are multiplied by normalizing factors called gains. There are three gains in the fuzzy controller, for du, dy and C, respectively. Here C is the notation for *next-change-in-cores*, i.e., $du(k+1)$ in Figure 4.2. The gains are denoted by g_{du}, g_{dy}

and g_C, and have a significant impact on the system performance. For example, if g_{dy} is 10, *change-in-throughput* is one hundred percent considered as "increase_large" when the measured change is larger than 20% (see Figure 4.3(b)). If g_{dy} is 20, the threshold of one hundred percent "increase_large" becomes 10%, and the controller will be more sensitive to *change-in-throughput*. Besides the three normalizing gains, the control interval is also an important parameter. With a shorter control interval, the system responds more quickly, but the system may become more fluctuating.

The final output of *next-change-in-cores* is the combination of the consequents of all the involved rules. The weighted average method, i.e., the AVG function in Algorithm 4, is used for defuzzification. The equation to calculate the final output value is

$$u^* = \frac{\sum \mu_C(u) \cdot u}{\sum \mu_C(u)} \tag{4.2}$$

where u is the centre value of each membership function in Figure 4.3(c) ($\pm 1, \pm 2$), and $\mu_C(u)$ is the respective certainty of each consequent.

Last, a complete example is given to walk through the whole fuzzy-control algorithm. Assume that after one more core is allocated to an application, the measured UIPT increases by 17.5% compared to the last measurement, i.e., $du = 1$, $dy = 0.175$. The three gain factors are $gdu = 1$, $gdy = 10$, and $gdC = 1$. Based on the two membership functions shown in Figure 4.3(a) and Figure 4.3(b):

$$du * gdu = 1 \Longrightarrow \mu_{cores-increase} = 1.0, \mu_{cores-decrease} = 0$$

$$dy * gdy = 1.75 \Longrightarrow \mu_{UIPT-increase_large} = 0.75, \mu_{UIPT-increase} = 0.25$$

Based on the fuzzy rules in Table 4.1:

$$\mu_{cores-increase-next} = min(\mu_{cores-increase}, \mu_{UIPT-increase}) = 0.25$$

$$\mu_{cores-increase_large-next} = min(\mu_{cores-increase}, \mu_{UIPT-increase_large}) = 0.75$$

Based on the AVG function in Equation 4.2

$$dC = \frac{\mu_{cores-increase-next} * 1 + \mu_{cores-increase_large-next} * 2}{\mu_{cores-increase-next} + \mu_{cores-increase_large-next}} = 1.75$$

where 1 and 2 are the centre values in Figure 4.3(c).

When the core allocation is an integer, the final output is $round(dC * gdC) = 2$, so two more cores will be allocated to this application in the next interval. This example shows how the controller determines the CPU allocation based on two inference results with different certainties. Because the change in UIPT is significant (0.175 or 17.5%), the controller will use a large search step in the next interval. By comparison, the hill-climbing algorithm in the previous section uses the same search step in different situations.

4.4 Evaluation of Allocation Algorithms

This section presents an experimental study that evaluates the two allocation algorithms. Four types of workloads with different characteristics are generated to examine how the algorithms perform with different performance curves and workload changes. There is no standard benchmark for adaptive CPU allocation in the literature, so this research compares the experiment results to the best results with static allocations. In the experiments, the algorithms can find the best CPU allocation for stable workloads, and adjust the allocations with respect to the requirements of dynamic workloads.

The Linux machine introduced in Section 3.4 is used for experiments in this section. The allocation program allocates and isolates CPU resources via Linux cgroups. If a client program is used to generate requests to a server application, this client program is assigned to a separate set of CPU cores. The allocation algorithms do not rely on any particular way to allocate CPU resources. Both partitioning and time-based allocation mechanisms are tested in the following experiments. The core partitioning is based on the cpuset subsystem of Linux cgroups, while the time-based allocation is based on the quota mechanism in the cpu subsystem.

The algorithms use the same parameters throughout all the experiments in this section. The configuration is shown in Table 4.2 (see Algorithm 2 and Algorithm 3 for the meaning of these parameters). These parameters are not determined through fine-tuning and may not be optimal for each workload. The objective of the experiments using these parameters is to evaluate the feasibility of the overall methodology. Evaluation results show that these values are robust for a wide range of workloads. Different workloads are not very sensitive to the change thresholds in the two algorithms. The measurement interval affects measurement noise and controller stability. When the interval is too short, the target workload may also suffer, because frequent changes to the CPU set cause execution with cold caches. Finding the optimal parameters for a given workload is another interesting problem and should be considered in future work.

Table 4.2: Algorithm Parameters

Common Parameters	
measurement interval	1 sec
greedy core reduction threshold	-0.01
demand change detection threshold	0.5
Hill Climbing Parameters	
change threshold	0.04
Fuzzy Control Parameters	
gain factor for relative change in UIPT (gdy)	12
gain factor for previous change in cores (gdu)	1
gain factor for change in core allocation (gdC)	1
Multi-interval Corrector Parameters	
CPU adjustment	2 cores
multi-interval window	3 intervals
change threshold	0.1

4.4.1 Stable Workload

The first experiment in this section examines whether the multi-timescale hill-climbing algorithm can find a proper CPU allocation for a stable workload. The TPC-E workload used in Section 3.4 is used in this experiment. The same MariaDB server and system configurations are used. The server is warmed up using this workload before the experiments. The client program has 50 concurrent threads to generate requests. For each client thread, requests are sent to the database server immediately when the previous one is finished. With a static CPU allocation, the throughput of the database server does not fluctuate because similar transactions are being processed. With core partitioning, this workload is a stable workload with a concave performance curve. The best static CPU allocation for this workload is 18 cores (see Figure 3.3 on Page 45).

In this experiment, the algorithm is executed repeatedly with two different initial core allocations, 20 times each. One is lower than the ideal static allocation, while the other one is higher. Each run lasts for 180 seconds to observe whether the allocation algorithm can reach the ideal core allocation (18 cores) and how stable the algorithm is after reaching the ideal core allocation.

Figure 4.4 illustrates the detailed core allocations during two sample experiment runs. Without any prior knowledge about the optimal number of cores, the hill-climbing algorithm can converge toward the ideal core allocation regardless of the initial core allocation.

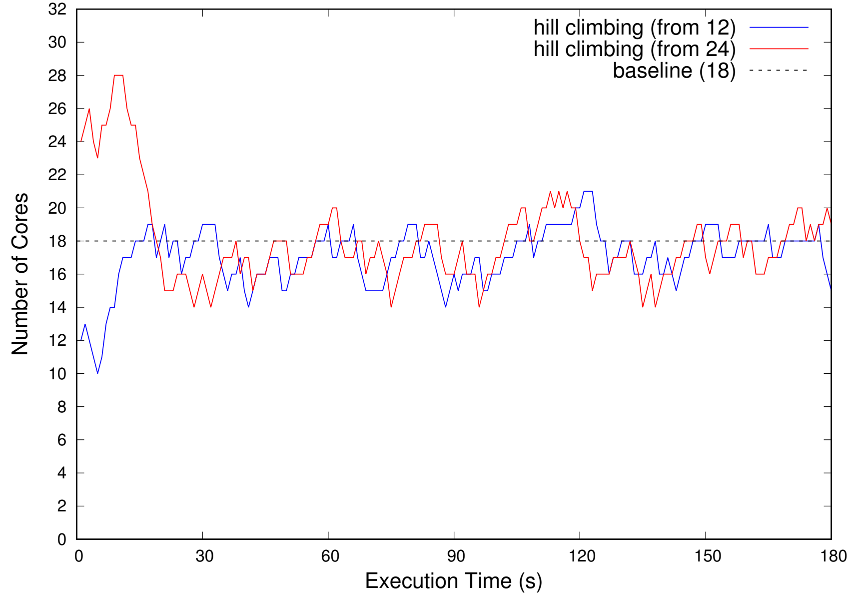


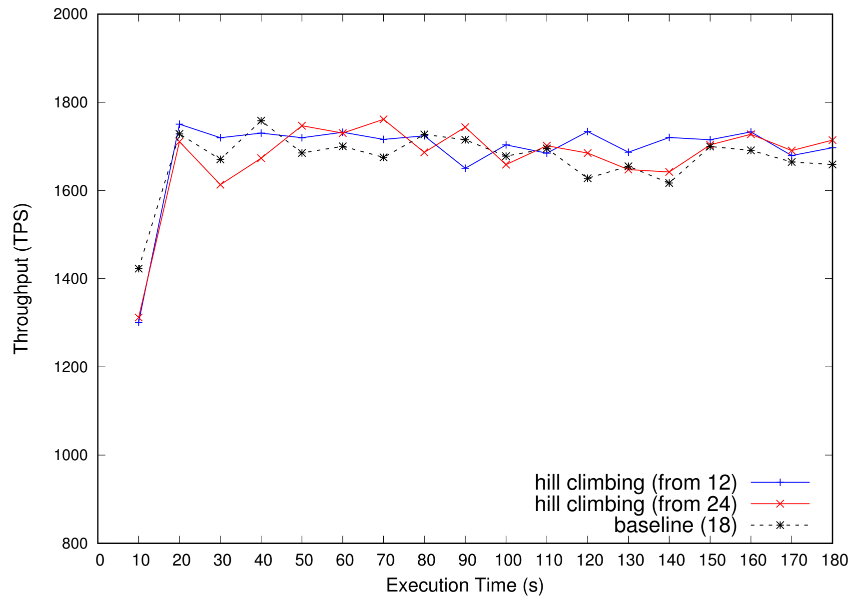
Figure 4.4: Core Allocation with Hill Climbing for Stable TPC-E Workload

The time for the algorithm to reach the ideal allocation depends on the initial core allocation. When starting from 12 cores, the convergence takes 8.33 intervals on average. In contrast, it takes longer to search from the other side. When starting from 24 cores, the average time is 17.7 intervals.

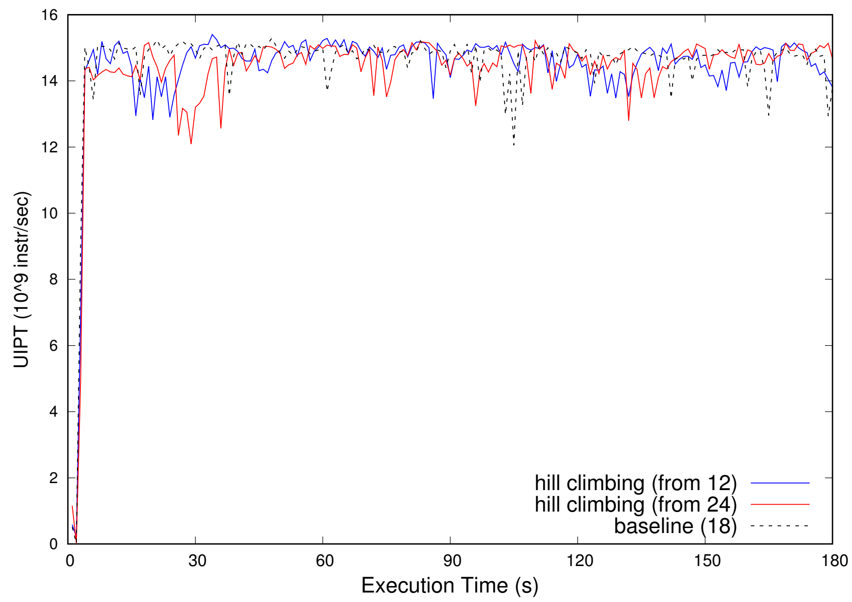
After convergence, the performance of the hill-climbing algorithm is stable. This study considers the average number of cores after convergence in each run as a sample mean. When the initial allocation is 12 cores, the mean of 20 sample means is 17.26, and the coefficient of variation is 0.01. When the initial allocation is 24 cores, the mean of 20 sample means is 17.30, and the coefficient of variation is 0.02. In the following experiments, the average number of cores after convergence is also computed in this way.

The measurements of TPS and UIPT during the same experiment runs are depicted in Figure 4.5. The baseline measurements with 18 cores are also plotted in the same figure as a reference. The hill-climbing algorithm achieves performance close to the baseline at both the application level and the instruction level. The average throughput of 40 runs reaches 97.4% of the baseline performance. The coefficient of variation is only 0.03, so this algorithm performs consistently for the stable TPC-E workload.

As a comparison, the same experiment is repeated with the fuzzy-control algorithm. The average throughput is 96.8% of the baseline result. The coefficient of variation of the



(a) Throughput Result for TPC-E Workload



(b) UIPT Changes for TPC-E Workload

Figure 4.5: Results with Hill Climbing for Stable TPC-E Workload

throughput result is 0.03. The average numbers of cores after convergence with the two different initial allocations are 17.26 and 17.40 respectively. The coefficients of variation in the two cases are both less than 0.02. Because the results with the two algorithms are quite close, the detailed results with the fuzzy-control algorithm in this and following experiments are presented in Appendix A.

4.4.2 Unsaturated Workload

In the second experiment, the two algorithms are evaluated using an unsaturated workload. When CPU allocation is higher than the requirement of this workload, CPU cores do not always find threads to run, and the application performance almost does not change. Thereby the performance curve is not downward concave as in the first experiment, but contains a plateau. Moreover, this experiment shows that the algorithms in this work are not specific to database systems and core partitioning.

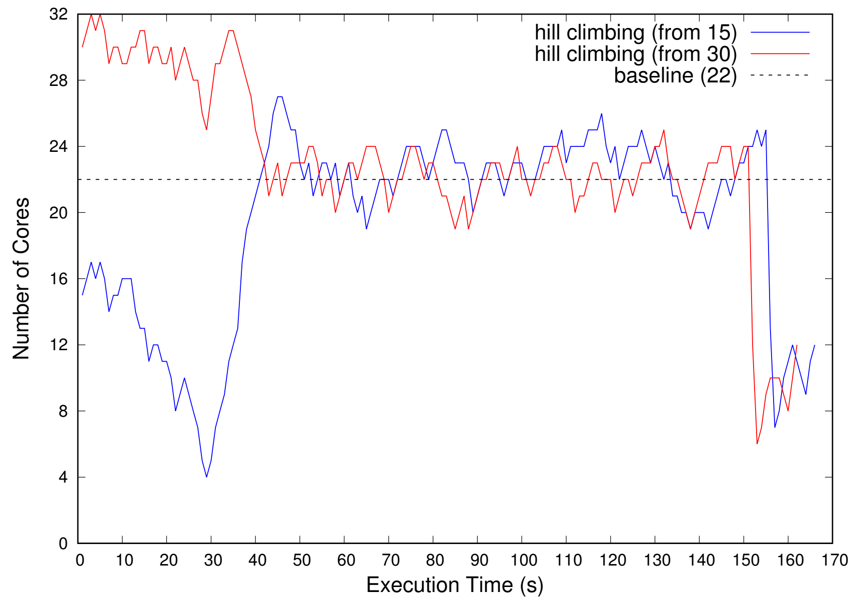
The Canneal benchmark program from the Parsec suite [6] is chosen for this experiment. This Canneal program employs cache-aware simulated annealing to minimize the routing cost of a chip design. Different from database systems that use locks for synchronization, Canneal uses fine-grained parallelism with a lock-free algorithm. Its aggressive synchronization strategy is based on data race recovery instead of avoidance. The performance metric of interest for this program is execution time. In this experiment, the number of threads in the program is set to 64, the number of "temperature steps" in the Canneal algorithm is set to 30,000, and the native input data set is used.

The quota mechanism in Linux cgroups is used to allocate CPU resources in this experiment. The Canneal program can access 32 physical cores but with a time limit. The allocation program in this experiment adjusts CPU allocations by changing the quota value of effective cores. For this workload, the optimal static CPU allocation is 22 effective cores. When the quota of CPU time is equivalent to 22 effective cores or more, the execution time of this workload almost does not decrease. Similar to the first experiment, two different initial quota values are tested in this experiment, 15 and 30 effective cores.

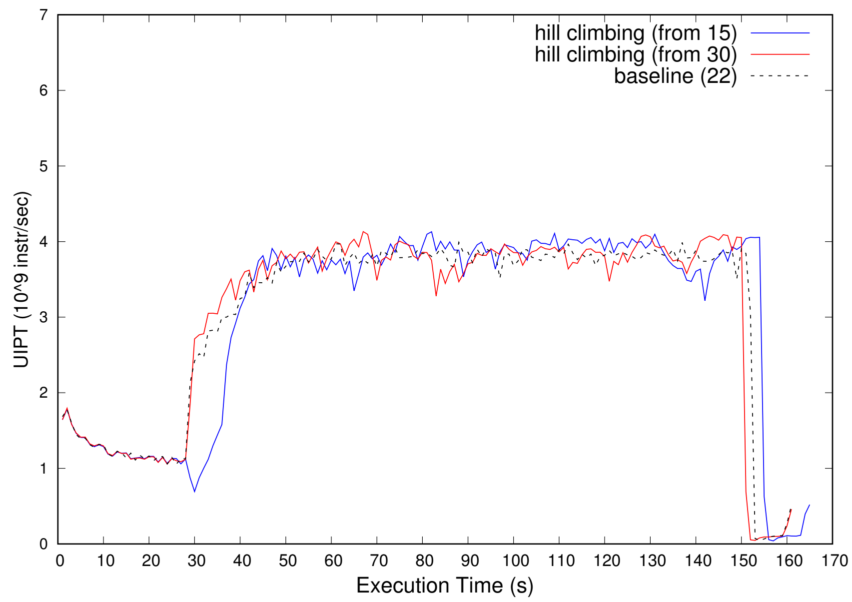
Figure 4.6(a) shows that after the CPU allocation reaches the ideal allocation of 22 effective cores and the hill-climbing algorithm keeps a steady CPU allocation until the Canneal program finishes. The average quota value of 20 runs with 15 and 30 initial cores are 21.87 and 22.07 respectively, very close to the ideal allocation. The coefficient of variation in either case is less than 0.03.

Figure 4.6(b) shows the execution time and UIPT measurements with the hill-climbing algorithm. The baseline measurements with 22 effective cores are also plotted in this figure.

The average execution time of 20 runs is 5% longer than the baseline execution time when the initial allocation is 15 effective cores. When the initial allocation is 30 effective cores, the average execution time of 20 runs is only 2% longer than the same baseline result. In both the cases, the hill-climbing algorithm still performs consistently. The coefficient of variation of the average execution time is less than 0.02 in either case. It can be seen from Figure 4.6(b) that the offered load is low during the first 28 seconds. Therefore, the allocation algorithm keeps reducing the quota value (refer to Figure 4.6(a)). When the in-



(a) Core Allocation for Canneal Benchmark



(b) UIPT Changes for Canneal Benchmark

Figure 4.6: Results with Hill Climbing for Canneal Benchmark

itial allocation is 15 effective cores, the allocation is gradually reduced to a small number. When the workload increases later, it takes more time for the algorithm to increase the CPU allocation to the ideal level compared to the experiment starting with 30 effective cores. That is why the average execution time is about 3% longer when the initial allocation is 15 effective cores. In addition, the AIMD mechanism (see Algorithm 2) is triggered at the beginning and end of the experiment because of the substantial changes in workload.

The fuzzy-control algorithm performs as well as the hill-climbing algorithm in this experiment. When the initial allocation is 15 effective cores, the average execution time is 6% longer than baseline result, and the average quota value after convergence is 20.67. When the initial allocation is 30 effective cores, the average execution time is only 2% longer than the baseline result, and the average quota value after convergence is 20.68. In the two cases, the coefficients of variation of the execution time are less than 0.01. The coefficients of variation of the quota value are not greater than 0.04. The details are shown in Appendix A.

4.4.3 High-Frequency Dynamic Workload

A workload in practice may have highly frequent changes which bring interference to a feedback-based algorithm. The multi-timescale mechanism in this work is used to overcome this kind of interference. The third experiment validates this design by applying the two algorithms to such a workload.

The workload in this experiment is based on the TPC-E workload used above. Each client thread implements a think-time mechanism between requests to vary the offered load for the server. Each think time is a random integer between 0 and 200 ms generated according to a uniform distribution, and thus the mean is 100 ms. The workload changes can be measured at the frequency of feedback measurements. The magnitude of the consequent performance changes is close to the performance change arising from a change in CPU allocation, and thus affects the judgment of the algorithms. This workload is also unsaturated because CPU cores do not always have work to do. Partitioning allocation is used in this experiment. The best static allocation for this workload is 7 cores. With more than 7 cores, the performance improvement of this workload is negligible.

First, the criticality of the multi-timescale mechanism is verified by confirming that the single-timescale hill-climbing algorithm in Algorithm 1 does not reliably converge to the ideal allocation for this high-frequency dynamic workload. Figure 4.7 shows the core allocation in two sample runs with this algorithm when the initial allocation is 3 or 12 cores. When the initial allocation is 3 cores, the algorithm finds the optimal allocation

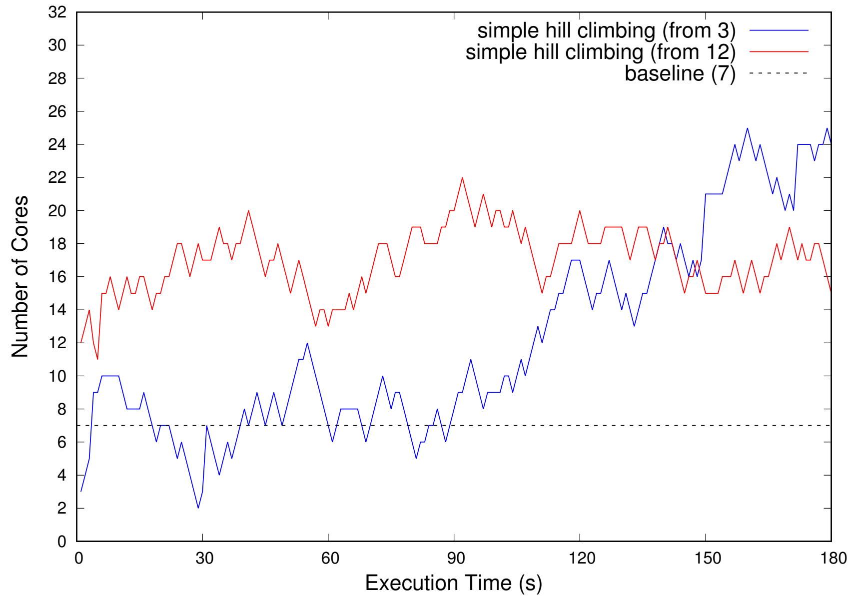


Figure 4.7: Core Allocation with Single-Timescale Hill Climbing
for Stable TPC-E Workload

in the first place but is misled to a wrong range later. When the initial is 12 cores, the algorithm continuously searches in a wrong range. With only the single-timescale hill-climbing algorithm, the core allocation does not have a stable pattern. The experiment is repeated 20 times with two different initial allocations respectively. The results shown in the figure are two examples. When the initial allocation is 3, the coefficient of variation of the average core allocation is 0.38. With an initial allocation of 20 cores, the coefficient of variation is 0.46.

In contrast, the hill-climbing algorithm with the multi-timescale mechanism controls the CPU allocation within a reasonable range even when a total of 32 cores are provided. Figure 4.8 shows the detailed CPU allocation in two sample runs with different initial core allocations. When the initial allocation is 3 cores, after the algorithm finds the proper allocation, the average number of cores is 6.71. Each run takes 180 measurement intervals, and the CPU allocation never deviates from the baseline too much. More importantly, when the initial allocation is more than the ideal 7 cores, the algorithm overcomes the interference of highly frequent workload changes, and successfully reduces the allocation to a reasonable level. In this case, the average core allocation after the convergence is 6.72. The coefficient of variation of the average core allocation is larger than that in other experiments, reaching about 0.14 with either 3 or 12 cores. This is because the mean of

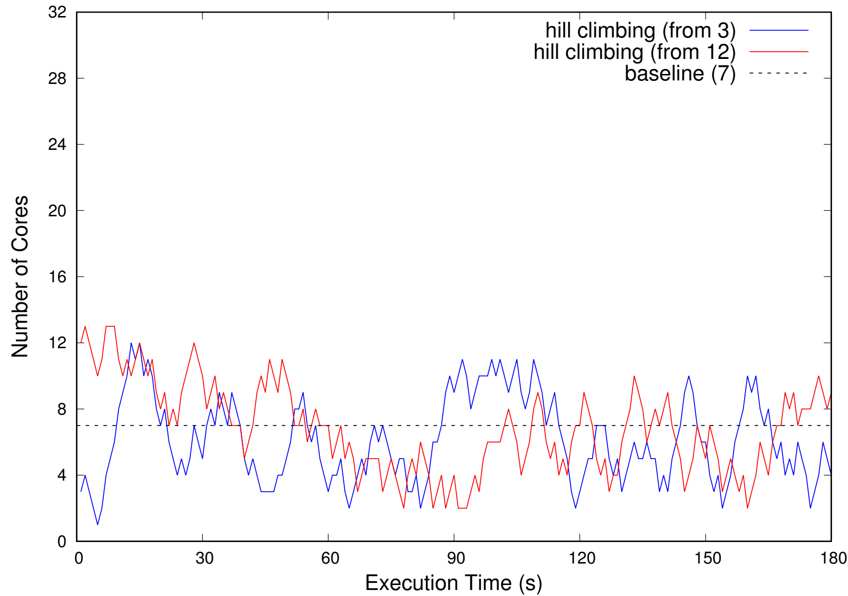
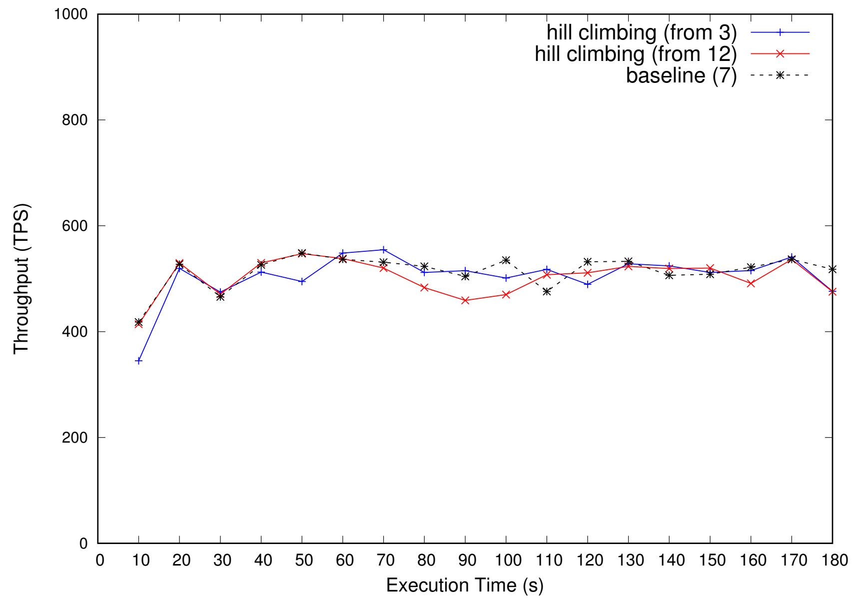


Figure 4.8: Core Allocation with Hill Climbing for High-Frequency Dynamic TPC-E Workload

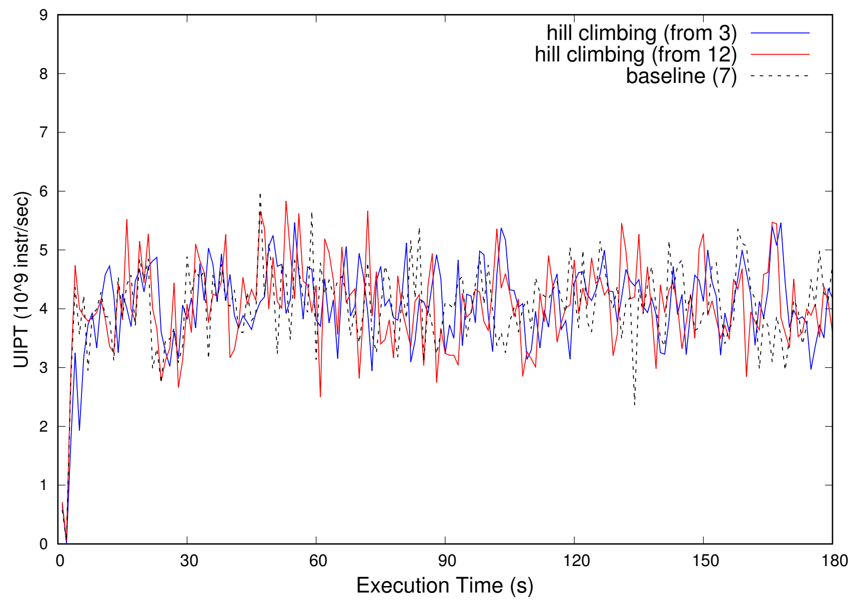
core allocation is not a large number. The actual core allocation in each run is not much different from each other.

Figure 4.9 presents the TPS and UIPT measurements with the hill-climbing algorithm during two experiment runs with different initial allocations, 3 and 12 cores. The mean TPS of 20 runs in the two cases both reach 97% of the baseline result with 7 cores, and the coefficient of variation is only 0.01. Figure 4.9(b) illustrates that even with a static CPU allocation, small changes in UIPT are constantly measured. The multi-interval corrector in the algorithm design is required to eliminate the interference from these fluctuations.

Similarly, the fuzzy-control algorithm achieves 95% of the baseline throughput with an initial allocation of 3 or 12 cores. The average numbers of cores after convergence are 7.16 and 7.25 respectively in the two cases. The multi-timescale mechanism still plays an important role in dealing with high-frequency workload changes. The details are shown in Appendix A.



(a) Throughput Result for High-Frequency Dynamic TPC-E Workload



(b) UIPT Changes for High-Frequency Dynamic TPC-E Workload

Figure 4.9: Results with Hill Climbing for High-Frequency Dynamic TPC-E Workload

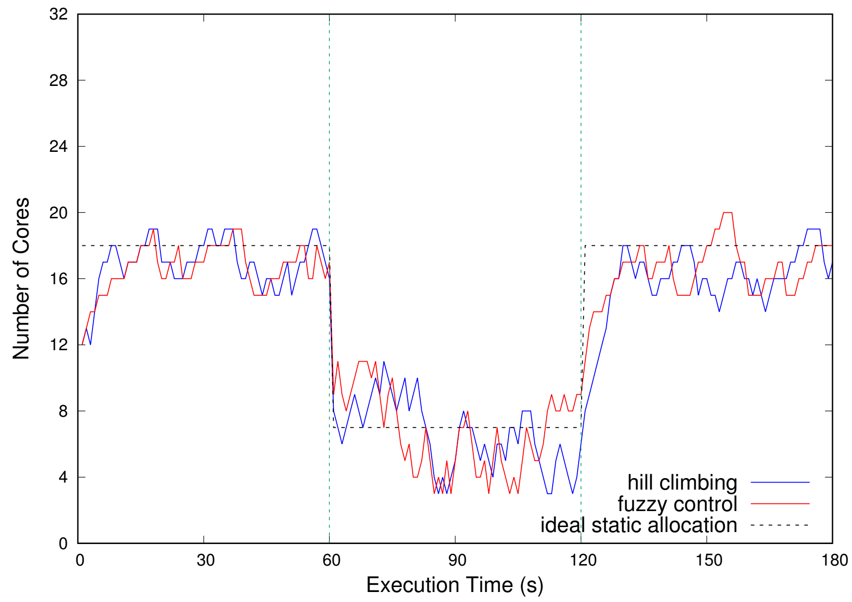
4.4.4 Low-Frequency and Large-Magnitude Dynamic Workload

The last experiment examines whether the algorithms can adapt to a dynamic workload with infrequent substantial changes, and find the respective optimal static allocations. Two TPC-E workloads with different think times are combined to generate a dynamic workload in this experiment. This dynamic workload is changed from high-demand to low-demand and back to high-demand alternately with each phase being 60 seconds. The high-demand phase is identical to the stable workload with no think time. The low-demand phase is the high-frequency dynamic workload with an average think time of 100 ms. Compared to the measurement frequency, the change between the two workloads does not happen frequently, but the magnitude is large.

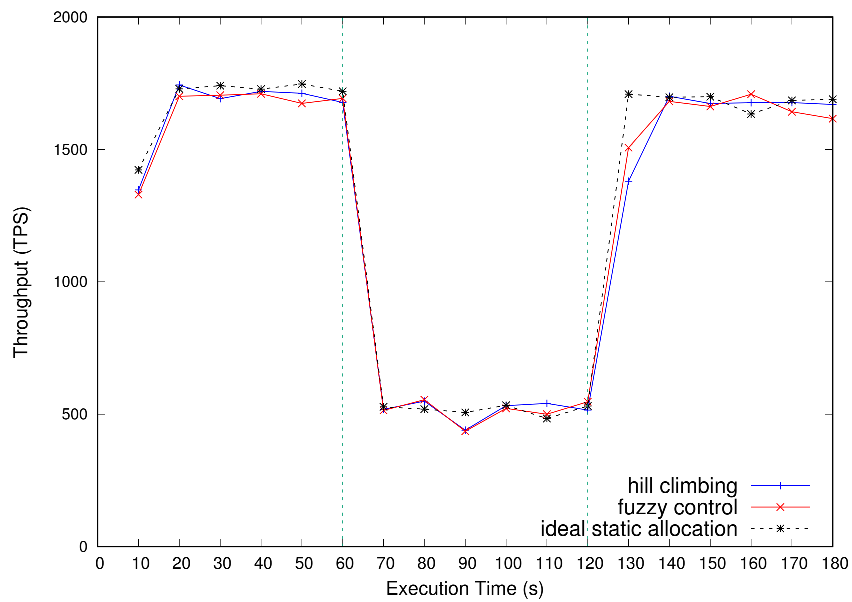
The changes in core allocation and TPS in Figure 4.10 show how the two algorithms perform during the whole 180-second experiment. Initially 12 cores are allocated to the database server. The vertical dashed lines indicate when the workload changes. Figure 4.10(a) shows the detailed core allocations controlled by the two algorithms. What can be clearly seen from this figure is that the algorithms perform very well in each phase. Moreover, both the algorithms react quickly to the workload changes. The AIMD mechanism speeds up the adjustments when the workload changes dramatically.

Figure 4.10(b) shows the throughput results with the two algorithms. As a comparison, Figure 4.10(b) includes a synthetic but unreachable optimum result comprised of the throughput observed with respective optimal static allocations, i.e., 18 cores for the high-demand phase and 7 cores for the low-demand phase.

The performance of algorithms is evaluated by an average score through the high-demand and low-demand phases. First, a score in each phase is calculated respectively by dividing the average TPS using the optimal result of that phase. The overall score through the two phases is the mean of the two scores in each phase. The same experiment is repeated 20 times. For both the hill-climbing and fuzzy-control algorithms, the overall score is 0.98 on average compared to the baseline optimal result. For either algorithm, the coefficient of variation of the throughput result is less than 0.02. To sum up, this experiment confirms that the allocation algorithms can automatically adapt to a dynamic workload with substantial changes and deliver performance that is close to the theoretical (but unattainable) optimum.



(a) Core Allocations for Low-Frequency Dynamic TPC-E Workload



(b) Throughput Results for Low-Frequency Dynamic TPC-E Workload

Figure 4.10: Results for Low-Frequency Dynamic TPC-E Workload

Chapter 5

CPU Resource Manager for Multiple Tenants

This chapter studies how to dynamically share CPU resources among multiple tenants in a work-conserving manner and enforce resource isolation at the same time. The multi-tenancy in the resource sharing context means multiple workloads share the same resources. The UIPT metric and the adaptive allocation algorithms described in the previous chapters provide the foundation for the resource manager presented in this chapter. The UIPT metric enables the resource manager to handle various applications regardless of their specific performance metrics. The adaptive allocation algorithms are responsible for detecting the CPU requirement of each tenant workload. In this thesis, the CPU requirement of a tenant refers to the CPU resources required for its software system to achieve the optimal performance.

This chapter presents a resource manager that allocates CPU resources to multiple tenants based on predefined sharing policies and tenant requirements. The sharing policy employed by the resource manager determines the theoretical share of each tenant. When a tenant's CPU requirement is less than its theoretical share, the resource manager ensures that this tenant's requirement is met. When a tenant's requirement is more than its theoretical share, the resource manager ensures that this tenant can consume at least its theoretical share of CPU resources. When multiple tenants require CPU resources in excess of their theoretical shares, and there are no idle CPU resources, the resource manager enforces resource isolation and ensures that no tenant is degraded because its CPU share is occupied by other tenants. Meanwhile, the resource manager is work-conserving, so when there are otherwise idle CPU resources, a tenant can consume more CPU resources than its theoretical share subject to a certain policy.

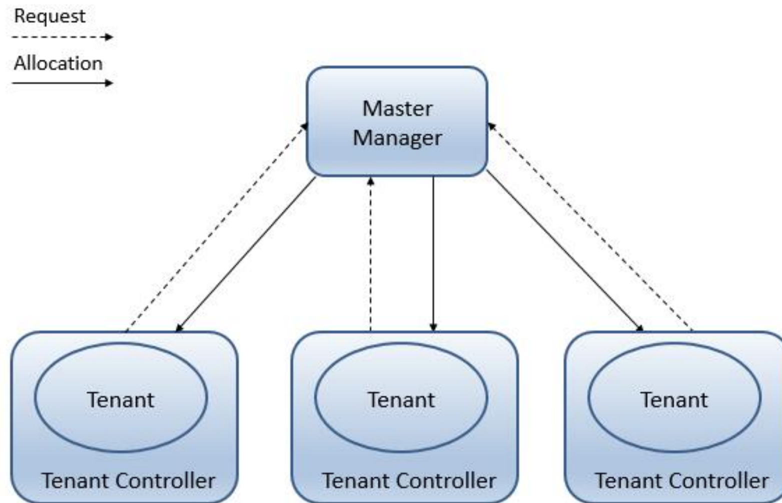


Figure 5.1: Resource Manager Architecture

The resource manager supports various sharing policies and is extensible. A prototype of the resource manager is implemented with four different sharing policies: proportional sharing, strict priority, guarantee, and efficiency. This chapter first presents the overall architecture of the CPU resource manager, and then introduces how the four policies are implemented based on feedback measurements. The last section shows the evaluation results regarding these sharing policies.

5.1 System Overview

The resource manager uses the two-level architecture shown in Figure 5.1. Each tenant controller measures UIPT changes of a tenant workload, determines its resource requirements, and reports resource requests to the master manager periodically. The master manager makes the allocation decision after collecting resource requests from all the tenant controllers. Then the master manager adjusts the CPU allocation to each tenant workload and notifies their tenant controllers accordingly. The resource manager is responsible for managing CPU resources on a single machine, so it is a concurrent system, but not a distributed system.

A tenant controller determines the CPU requirement of a workload based on recent CPU allocations and consequential performance changes (see Section 4.1). Application

performance is estimated via UIPT. The tenant controller can use either of the algorithms introduced in Chapter 4. A perennial weakness of this kind of feedback-based system is that the system reaction may lag when the workload changes. The resource manager reduces the impact of this disadvantage by using UIPT. UIPT is a low-level and generic metric that can be measured in sub-seconds, so the adjustment of resource allocation can be done in a short time interval. Another issue of previous feedback-based resource management systems is that when application-level performance is measured by task completion time, there is no appropriate metric for online feedback measurement. The resource manager presented here does not have this problem, because UIPT is a good proxy of application performance even while the workload is still running.

When the available resources are not sufficient to meet all requests from tenant controllers, the master manager must arbitrate between competing requests based on a sharing policy. Tenants that share a set of resources must also adopt the same sharing policy. For example, with a proportional sharing policy, each tenant can obtain only a fraction of the total available resources proportional to its weight; with a strict priority policy, the tenant with the highest priority has its CPU requirement met first. The resource manager is able to support multiple different sharing policies.

A prototype of the resource manager is implemented using C++. The prototype operates independently from the application software and workload-specific performance metrics, and does not require modifications in the OS kernel or tenant applications. In the prototype, each tenant controller is a separate thread and the main process works as the master manager. The prototype relies on Linux-specific interfaces. First, UIPT is measured via Linux system calls for performance monitoring. Moreover, in the prototype, CPU resources are allocated using the partitioning or the quota mechanism in Linux cgroups. The partitioning mechanism refers to the *cpuset* subsystem of Linux cgroups. The quota mechanism, which is in the *cpu* subsystem, sets an upper limit on the CPU time that tasks in a control group can use. Each tenant application is placed in a sub-group, and the group name is an input to the resource manager. The prototype does not require users to specify CPU requirements for their applications. CPU allocation is adjusted automatically according to tenant requirements. The evaluation experiments in this chapter are based on the hill-climbing algorithm in Section 4.2. Three commonly used sharing policies and one novel policy are evaluated with this prototype.

When using core partitioning, the resource manager allocates at least one CPU core to each tenant for two reasons: one, the prototype must keep each tenant application running and measure UIPT changes; two, the parameter in the cgroup configuration cannot be blank. When using the quota mechanism, the allocation granularity can be less than one core, but the prototype still needs to keep a minimum amount of CPU resources for each

tenant. This minimum allocation may violate the sharing policy in certain cases. For example, when two workloads share a total of 16 cores under the priority policy, and the high-priority workload requires all 16 cores, it can obtain only 15 cores. This is a limitation of the resource manager.

5.2 Sharing Policies

5.2.1 Proportional CPU Sharing

With proportional CPU sharing, every tenant has a weight and receives a theoretical share of the available resources proportional to its weight. With a work-conserving resource management system, a tenant is allowed to use more resources than its theoretical share. The proportional sharing policy defines how idle CPU resources should be shared among tenants that have requirements in excess of their theoretical shares, i.e., still proportional to their weights.

The resource manager in this thesis can use core partitioning or the quota mechanism to restrict each tenant from consuming more CPU resources than its fraction when there is CPU contention, and adjusts the limit adaptively. In particular, tenant controllers determine CPU requests based on the hill-climbing algorithm in Chapter 4, and the master manager employs a weighted max-min fairness algorithm [38] to determine CPU allocation among tenants.

The algorithm shown in Algorithm 5 maximizes the minimum share of a tenant whose demand is not fully satisfied. First of all, the fair share unit is calculated using the current total weight to divide the total residual CPU resources (Line 13 in Algorithm 5). In each iteration, the quantity of CPU resources a tenant can obtain is its weight times the share unit. If the tenant does not require that many CPU resources, only the quantity specified in the request is allocated to the tenant (Lines 15 - 19 in Algorithm 5). The residual amount is allocated in the next iteration to the remaining unsatisfied tenants until there is no residual resource or unsatisfied tenant (Lines 3, 10 - 12 in Algorithm 5). If core partitioning is used for CPU allocation, the CPU cores are considered as indivisible, so the allocation numbers returned by this algorithm need to be rounded to the nearest integer.

Algorithm 5 WeightedMaxMin(requests)

Configuration:
total: the total available CPU resources
tn: the number of tenants
weights: the array of weights of tenants
Input:
requests: the array of CPU requests from tenant controllers
Output:
allocations: the array of CPU allocations to tenants

```
1: available ← total
2: allocations[1 .. tn] ← 0
3: while available > 0 do
4:     sum_weight ← 0
5:     for t ← 1 to tn do
6:         if allocations[t] < requests[t] then
7:             sum_weight ← sum_weight + weights[t]
8:         end if
9:     end for
10:    if sum_weight = 0 then
11:        break while
12:    end if
13:    unit ← available/sum_weight
14:    for t ← 1 to tn do
15:        if allocations[t] < requests[t] then
16:            delta ← Min(unit * weight[t], request[t] − allocation[t])
17:            allocations[t] ← allocations[t] + delta
18:            available ← available − delta
19:        end if
20:    end for
21: end while
22: return allocations
```

5.2.2 Priority-Based CPU Sharing

Another fundamental sharing policies in practice is priority-based allocation among multiple tenant applications. With this policy, the tenant with the highest priority can obtain a CPU allocation equal to its requirement unless the available resources are not sufficient. Provided there are still enough resources remaining, the next highest priority tenant is able to have its requirement met, and so on.

With this sharing policy, the goal of CPU allocation is to guarantee the best-possible performance for high-priority workloads, while improving the performance of low-priority workloads as much as possible, corresponding to resource isolation and work conservation respectively. A typical example scenario is given by a database system tasked to process online transactional requests in near-realtime with high priority, such as account updates or purchasing requests. At the same time, analytical queries are also being executed at lower priority, but their completion time is still important for the overall outcome of the system operation.

The implementation of the priority policy is straightforward. Each tenant controller locally determines the CPU requirement of its tenant workload based on the feedback of UIPT changes. After that, the tenant controller reports the CPU requirement to the master manager as a resource request. Strict priorities are assigned to tenants and the master manager meets the requests of tenant controllers in the priority order. Thus, the requirement of low-priority workloads may be unfulfilled when the total CPU resources are not sufficient.

5.2.3 Resource Guarantee for Multiple Tenants

Another common sharing policy in practice is allocating a specific quantity of CPU resources to each tenant and guaranteeing that the tenant can use at least these CPU resources. With this policy, the guaranteed quantity is the theoretical share for each tenant when there is resource contention. Different from the proportional sharing or priority policy, the theoretical share with the guarantee policy is absolute, i.e., independent of other tenants. In reality, e.g., in a cloud environment, a virtual machine may be exclusively allocated a certain number of CPU cores to assure its quality of service. To deliver the guarantee, the CPU cores are partitioned, and all the tenant workloads need to be pinned to specific cores. Another common practice is setting an upper limit for the CPU time each tenant can use. These static mechanisms guarantee the quantity of CPU resources one tenant can utilize by restricting the maximum CPU resources that other tenants can utilize.

However, the static mechanisms are not work-conserving. If there are residual CPU resources, the work-conserving resource manager allows an unsatisfied tenant to utilize more CPU resources than its guaranteed quantity. Given that the guarantee policy does not define how residual CPU capacity is shared among unsatisfied tenants, another policy is required to determine the sharing, e.g., the proportional sharing or priority policy. When two policies are used together, the guarantee policy can be considered as a supplement to the other policy acting to ensure that the absolute minimum guaranteed resources are always present.

The implementation of this policy in the resource manager is based on the requests from tenant controllers. After receiving the requests, the master manager first subtracts the minimum value between each tenant's request and the guaranteed quantity from the total available CPU resources. Proportional sharing is used by the prototype together with the guarantee policy. The weighted max-min fairness algorithm in Algorithm 5 is used to share the remaining CPU resources among unsatisfied tenants. The guaranteed quantity of each tenant is employed as its weight in this algorithm.

5.2.4 Efficiency-Based CPU Sharing

In the first three policies, the CPU allocation among multiple tenants is based on each tenant's predefined weight, priority or guarantee instead of a metric describing the overall system. This section introduces a novel CPU sharing policy aiming to achieve the highest overall system efficiency.

The system efficiency is measured based on application performance. When the CPU allocation among multiple tenant applications changes, applications' performance may increase or decrease respectively. If the performance of different applications is normalized to a uniform score, the gains and losses can be compared. In this research, the CPU allocation that achieves the highest total normalized score is considered to be the most efficient allocation for the underlying infrastructure. In other words, the sharing policy in this section aims to obtain the best normalized performance from the whole system.

To support the efficiency-based sharing policy, the resource manager adjusts CPU allocation by comparing which tenant can obtain a higher (normalized) score with one more CPU core or equivalent quota time. At the application level, the normalized score is calculated as shown in Equation 5.1. In the resource manager, the baseline performance is the application performance with a single CPU core. This means that the efficiency policy intends to allocate CPU resources to the tenant application that is more scalable.

$$Normalized\ Score = \frac{Application\ Performance}{Baseline\ Performance} \times Efficiency\ Factor \qquad (5.1)$$

Moreover, because different tenants may have different merits for the infrastructure owner, an efficiency factor is used in the normalization. The efficiency factor is not the same as the weight in proportional sharing. In proportional sharing, each tenant's weight can be used to quantify its importance, but the proportional sharing policy does not take the runtime efficiency of applications into account. Allocating more CPU resources to the tenant with a higher weight does not always result in a better overall outcome.

For this efficiency policy, tenant controllers do not request CPU resources from the master manager. The adaptive allocation algorithms in Chapter 4 are not used here. Instead each tenant controller reports the change rate of UIPT with one more CPU core being allocated to its tenant application, and the master manager uses the change rates to estimate the score change. To calculate the change rate of UIPT, the change in UIPT when one core is added or reduced is divided by the average UIPT when the workload is executed with a single core. When the CPU allocation is changed, the UIPT measured in the current interval is compared with the previous average UIPT value. If the CPU allocation to a tenant does not change in a measurement interval, the measurement in this interval is used to calculate a new average UIPT of this tenant. The average UIPT with a single core needs to be measured in advance and passed to the tenant controller as an input.

The algorithm used in the master manager is shown in Algorithm 6. In every measurement interval, the master manager calculates the potential change of each tenant's normalized score with one more core based on the tenant's efficiency factor times the change rate of UIPT reported by its tenant controller (Line 7 in Algorithm 6). The change of each tenant's normalized score is pushed into a priority queue and the queue is sorted by the change values (Line 8 in Algorithm 6). The master manager first reduces one core from every tenant with a negative change value because these tenants will benefit from the reduction of CPU allocation (Lines 9 - 12 in Algorithm 6). Next if there are no idle CPU resources, it means that all the tenants have a positive change of normalized score. The resource manager takes one core from the tenant with the minimum change and gives the core to the one with the maximum change, because the gain is greater than the loss and the overall efficiency still improves (Lines 15 - 20 in Algorithm 6). While there are still remaining CPU resources, the master manager pops the next tenant with the maximum positive change from the queue and allocates one more core to this tenant (Lines 22 - 26 in Algorithm 6.

Algorithm 6 EfficiencyAllocate(rates)

total: the total available CPU resources
tn: the number of tenants
rates: the array of change rates of UIPT from tenant controllers
factors: the efficiency factors for tenants
allocations: the array of CPU allocations to tenants in last interval

```
1: available ← total
2: for t ← 1 to tn do
3:     available ← available − allocations[t]
4: end for
5: /* score_queue is the priority queue used to sort the score changes */
6: for t ← 1 to tn do
7:     score_change ← factors[t] ∗ rates[t]
8:     Push(score_queue, {score_change, t})
9:     if rates[t] < 0 then
10:        allocation[t] ← allocation[t] − 1
11:        available ← available + 1
12:    end if
13: end for
14: {max_change, max_index} ← PopMax(score_queue)
15: if available = 0 then
16:    if max_change > 0 then
17:        {min_change, min_index} ← GetMin(score_queue)
18:        allocation[min_index] ← allocation[min_index] − 1
19:        allocation[max_index] ← allocation[max_index] + 1
20:    end if
21: else
22:    while available > 0 AND max_change > 0 do
23:        available ← available − 1
24:        allocation[max_index] ← allocation[max_index] + 1
25:        {max_change, max_index} ← PopMax(score_queue)
26:    end while
27: end if
28: return allocations
```

There are different ways to calculate the normalized score based on application performance. This thesis presents only one of the meaningful possibilities. For example, the initial application performance when a tenant controller starts can also be used as the baseline performance in Equation 5.1 for normalization. The initial UIPT can be measured by each tenant controller directly. Therefore, tenant controllers do not require any input UIPT as a baseline value. If this is the case, the meaning of the normalized score is different. How to calculate the normalized score or how to define the overall efficiency depends on the concrete purpose of resource management.

5.3 Evaluation of Resource Manager

5.3.1 Experiment Design

The experiments in this section are used to evaluate the resource manager with respect to four different sharing policies. The policies are not specific to the resource manager. Thus, there are other ways to implement the policies. The resource manager is compared with various existing state-of-the-art mechanisms in terms of application performance and resource isolation.

For each policy, an experiment with two stable workloads is conducted to test whether the resource manager can enforce the sharing policy. In the experiments with stable workloads, there exists an ideal static allocation between two workloads. The purpose of the experiments is to test whether the resource manager can allocate CPU resources in the ideal way. Moreover, the best application performance with corresponding static allocation is measured as a reference to evaluate the application performance under the control of the resource manager. For the proportional-sharing and guarantee policies, the ideal static allocation is pre-specified. For the priority and efficiency policies, the ideal static allocation is based on application performance and can be found through preparatory experiments. The results with the current best-practice mechanisms that implement the same policy are also measured as a comparison.

Second, an experiment with a dynamic workload and a stable workload is used to verify the effectiveness of the resource manager in more complex scenarios. With a static allocation, CPU resources are not fully utilized if the requirements of workloads are time-varying. The experiment results with the resource manager are compared with the results with current best-practice mechanisms that implement the same sharing policies. The application performance is measured to show the gain resulting from the work-conserving feature of the resource manager.

Table 5.1: Current and Baseline Mechanisms for Different Policies

	proportional-sharing	priority	guarantee	efficiency
Static/Reference	static partitioning (cpuset), static usage limit (quota)	static partitioning (cpuset)	static usage limit (quota), weighted-fair (weight)	static usage limit (quota), weighted-fair (weight)
Best-practice	weighted-fair (weight)	emulated priority	static usage limit (quota)	N/A

Table 5.1 lists the static reference and best-practice mechanisms used in the experiments. The concrete mechanisms in Linux cgroups used in the experiments are indicated in the parentheses. The priority policy is usually emulated via weighted time-based allocation in reality. Since currently no other system supports the efficiency policy in practice, the best-practice mechanism in this case is not available.

In the evaluation of the proportional-sharing policy, both the partitioning mechanism and the quota mechanism in Linux cgroups are used by the prototype to allocate CPU resources. The resource manager is effective with different CPU allocation methods. Only the partitioning mechanism is used in the evaluation of the priority policy. With the partitioning mechanism, the primary workload in the experiment does not require all the cores to achieve its best performance. In contrast, with the quota mechanism, all CPU resources are required. (The performance curve is shown in Figure 2.1 on Page 21.) Hence the effectiveness of the resource manager in the priority scenario can only be shown when using partitioning. The quota mechanism is employed in the evaluation of guarantee and efficiency policies, since the quota mechanism is more generic than the partitioning mechanism. In the experiments, the allocation unit of quota time is an effective core. As in Section 2.2.1, the CPU time that is equivalent to one CPU core during a period of time is considered as an effective core.

The same machine and Linux system used in the previous chapters are used in the experiments. The concrete workloads and setups are introduced in the following subsections for each policy respectively.

In addition to the experiments in this chapter, the experiments in Section 2.2.2 are repeated using the resource manager proposed here. These experiments are used to compare resource isolation of different mechanisms. The results show that the resource manager

can enforce resource isolation in different proportional sharing scenarios as presented in Appendix B.

5.3.2 Proportional Sharing Policy

This section evaluates how the resource manager supports the proportional sharing policy. In Section 2.2.2, resource isolation of existing mechanisms with different weight ratios is studied. A weight ratio of 3:1 is chosen in the present section to verify that the resource manager can achieve resource isolation and work conservation simultaneously. Moreover, a representative CPU-intensive program is used to replace the cpuhog micro-benchmark used in Section 2.2.2. How the resource manager performs with other weight ratios and workload combinations is shown in Appendix B.

The first experiment in this section examines whether the resource manager can effectively isolate the CPU resources when there is CPU contention between two stable workloads. Specifically, in such a contention scenario, the resource manager needs to ensure that the fraction of resources that each tenant can utilize is proportional to its weight. Second, the resource manager is work-conserving, so a dynamic workload and a stable workload are executed together to verify the benefit of this feature.

A multithreaded benchmark program named swaptions from the Parsec benchmark suite [6] is used in this section. This benchmark is a CPU-intensive workload which uses the Heath-Jarrow-Morton (HJM) framework [32] to price a portfolio of swaptions. This program employs a Monte Carlo simulation to compute the prices. The stable and dynamic TPC-E workloads in Section 4.4 are co-located with the swaptions workload. In the stable workload, the clients submit transactions continuously with no think time. The dynamic workload executes multiple 120-second cycles. The 60 seconds high-demand phase uses the stable workload, while in the 60 seconds low-demand phase, each client inserts a random think time (0 to 200 ms) between transactions.

A total of 16 CPU cores are available in the experiments. In both the experiments, the weight of the static or dynamic TPC-E workload is 3, while the weight of the swaptions workload is 1. The experiments evaluate different versions of the resource manager that use two different mechanisms to allocate CPU resources (partitioning and quota). In addition, the application performance with static partitioning is measured, in which case the static or dynamic TPC-E workload is allocated 12 physical cores, while the swaptions program is allocated 4 cores. The quota mechanism in Linux cgroups is also tested as a reference. The static quota values for the two workloads are 1,200,000 and 400,000 microseconds respectively. Given the period of 100,000 microseconds in the cpu subsystem, these quota

values are equivalent to 12 and 4 effective cores. These static allocations follow the weight setting of 3:1.

Stable Workloads

The purpose of this experiment with two stable workloads is to test whether the resource manager can enforce the proportional sharing policy. The total 16 cores cannot meet the CPU requirements of the two workloads at the same time. Ideally, the TPC-E workload with a weight of 3 should obtain 12 cores, while the swaptions workload should obtain 4 cores. This experiment checks if the resource manager and the weight mechanism in Linux cgroups can allocate CPU resources in this ideal way. Because there is no way to directly measure the actual CPU allocation when using the weight mechanism in Linux cgroups, CPU usage of workloads is measured in this experiment instead. The perf_event interface in Linux [73] is used to measure the CPU usage of each workload during the experiments.

Figure 5.2(a) shows the CPU usage of the two workloads with different mechanisms. The measured CPU time is converted to the number of effective cores during the experiment and each result is the mean of 20 samples. The results with ideal static partitioning and static quota are also presented in this figure. It can be clearly seen that the CPU usage results with the resource manager is close to the results with static partitioning or static quota.

When core partitioning is used in the resource manager, the CPU usage of the TPC-E workload is 9.91 effective cores, while the CPU usage of the swaptions workload is 4.21 effective cores. When the resource manager controls the quota limits, the CPU usage results of the two workloads are 9.66 and 4.37 effective cores respectively. In contrast, with the weight mechanism in Linux cgroups, the CPU usage of the TPC-E workload is only 5.49 effective cores, whereas the CPU usage of the swaptions workload is 10.01 effective cores. Obviously, CPU time is not shared based on the 3:1 weight ratio with the weight mechanism, and this result is consistent with the findings in Section 2.2.2.

Figure 5.2(b) and Figure 5.2(c) show how the resource manager adjusts core partitioning or quota limits between stable workloads. At the beginning, the two workloads have equal resources. The resource manager quickly adjusts the CPU allocation according to the weight ratio and keeps a stable allocation afterwards. The average allocation to the TPC-E workload is close to 12 physical or effective cores, but because of the workload characteristics, the database server does not fully utilize the allocated CPU resources. Even with static partitioning, the TPC-E workload uses only 10.33 effective cores out of the 12 physical cores. With a static quota of 12 effective cores, CPU usage of the TPC-E

workload is only 10.02. The same experiments are repeated 20 times with the resource manager. For the average physical cores or quota time allocated to the TPC-E workload, the coefficient of variation is less than 0.02, showing that the resource manager perform consistently.

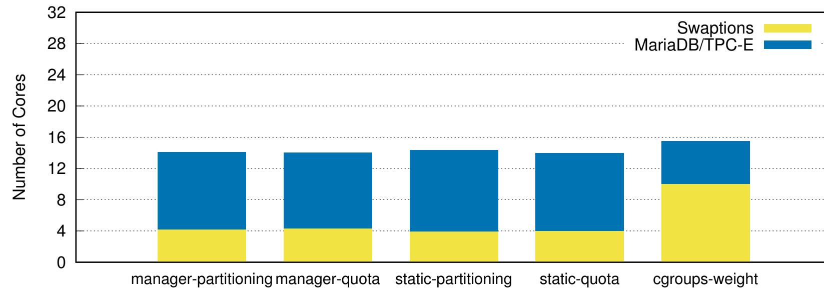
Figure 5.3 shows the throughput of the TPC-E workload with a weight of 3. When core partitioning is used, the average throughput of the TPC-E workload with the resource manager reaches 98.2% of the average throughput with static partitioning. When the quota mechanism is used, the average throughput of the TPC-E workload with the resource manager is 97.9% of the average throughput with static quota. This result confirms that the CPU allocation with the resource manager is close to the ideal allocation. Table 5.2 shows the execution time of the swaptions benchmark. With the weight mechanism, the execution time of the swaptions benchmark is shortest, because this benchmark workload consumes more CPU resources than its theoretical share.

The CPU usage of the two workloads with static quota is close to that with static partitioning. The execution time of the swaptions workload is also similar. Nevertheless the throughput of the TPC-E workload with a static quota is 15.1% lower than the throughput with static partitioning because of the excessive parallelism (see also Section 2.2.1). With the resource manager, the difference caused by the two mechanisms still exists.

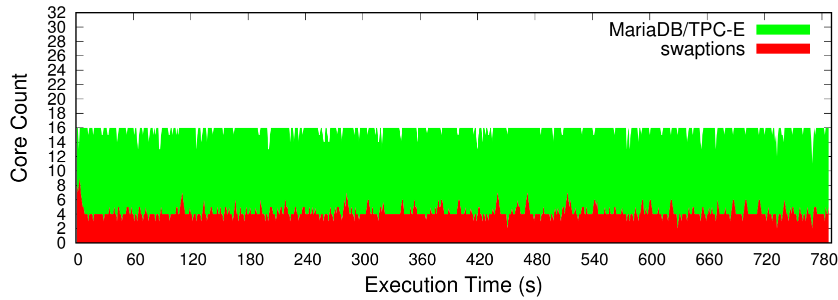
Dynamic Workloads

The experiment with the dynamic TPC-E workload and the stable swaptions workload is used to verify that the resource manager can maintain work conservation while enforcing resource isolation. The weights of the two workloads are still 3:1. The dynamic TPC-E workload does not require all the 12 cores in the low-demand phase, so the resource manager adaptively adjusts CPU allocation and allows the swaptions workload to use those otherwise idle CPU cores.

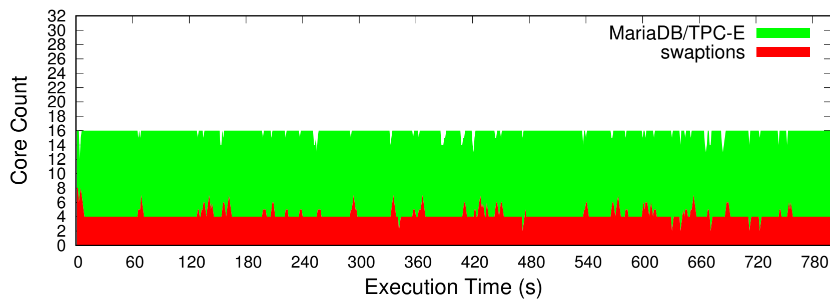
Figure 5.4 shows that the resource manager changes core allocation or quota limits as the TPC-E workload changes. When the CPU requirement of the TPC-E workload is high, and there is contention between the two workloads, the CPU allocation still follows the weight ratio of 3:1. When the requirement of the TPC-E workload decreases, the swaptions workload can consume more than its fraction (4 physical or effective cores). With 20 repeats of each experiment, the coefficient of variation of CPU allocation to the two workloads is less than 0.03 with either core partitioning or the quota mechanism. The resource manager stably achieves both resource isolation and work conservation.



(a) CPU Usage of Stable Workloads with Weights of 3:1



(b) Core Allocation with the Partitioning Mechanism



(c) Core Allocation with the Quota Mechanism

Figure 5.2: CPU Usage and Allocation between Stable Workloads with Weights of 3:1

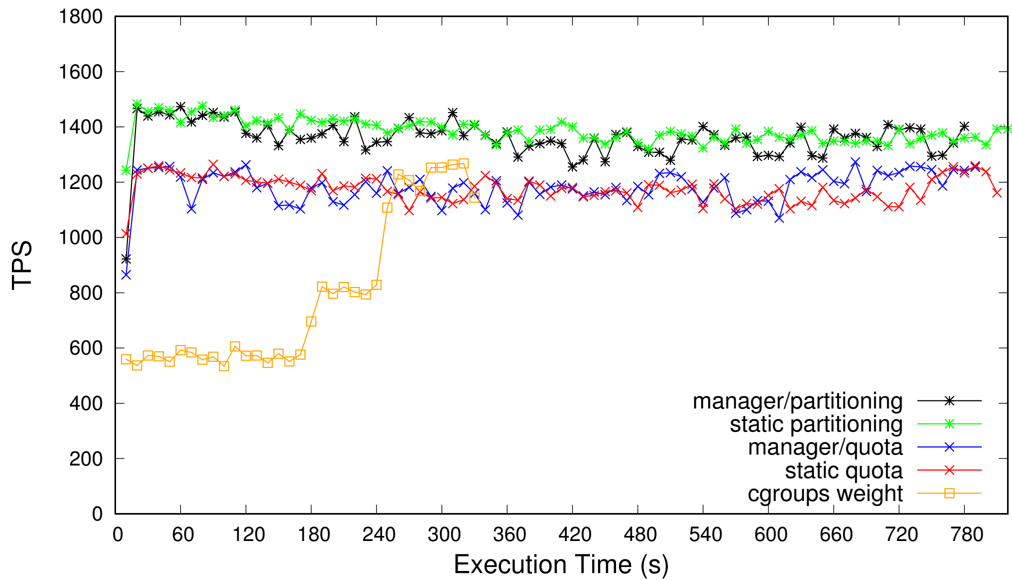


Figure 5.3: Throughput of MariaDB/TPC-E Workload

Table 5.2: Swaptions Execution Time with A Stable TPC-E Workload

	manager/partitioning	static partitioning	manager/quota	static quota	cgroups weight
Time	788.34s	825.63s	747.57s	819.15s	343.17s

Table 5.3 shows the completion time of the swaptions benchmark with different allocation mechanisms. With the resource manager, the dynamic partitioning result is 38.3% shorter than the result with static partitioning, and the dynamic quota result is 28.5% shorter than the result with static quota. The static allocation mechanisms are not work-conserving. The CPU resources are not fully utilized because the most CPU resources each workload can consume is restricted by physical cores or quota limits.

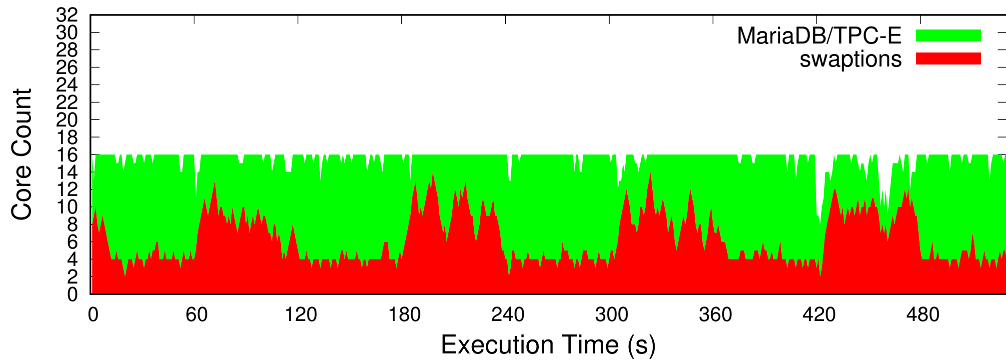
The weight mechanism is work-conserving, and thus the execution time of the swaptions benchmark with this mechanism is much shorter than other results. However, this mechanism cannot enforce the resource isolation based on weights. It can be seen from Figure 5.5 that, with the weight mechanism, the TPS result of the TPC-E workload declines significantly compared to the results in other cases, because the TPC-E workload cannot get its fraction of CPU resources even when its requirement is high. In contrast, the resource manager ensures that the CPU allocation follows the weight ratio when the requirement of the TPC-E workload is high. During the experiments, the TPS result with the resource manager is close to that with static partitioning or static quota. In both high-demand and low-demand phases, with either partitioning or quota, the average TPS with the resource manager is about 97% of the result with static allocation.

To sum up, the evaluation results show that the resource manager can achieve resource isolation and work conservation simultaneously with the proportional sharing policy. In contrast, static partitioning and static quota can enforce resource isolation but are not work-conserving. The weight mechanism in Linux cgroups is work-conserving but it does not ensure resource isolation. Moreover, the resource manager can work with both core partitioning and the quota mechanism.

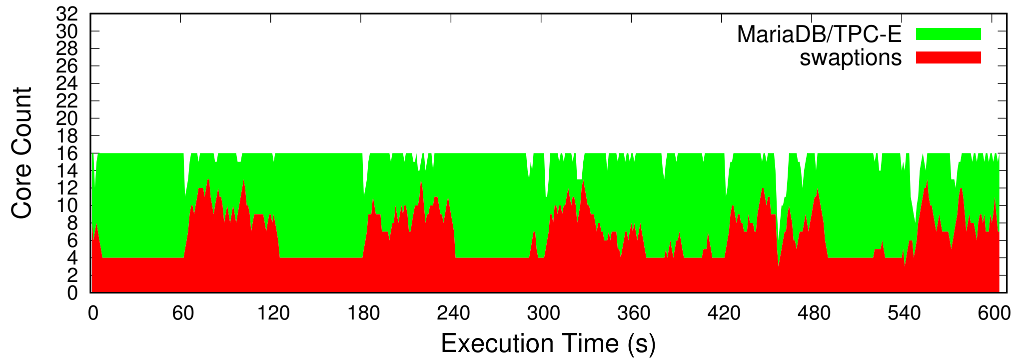
5.3.3 Priority Policy

The evaluation of the resource manager with respect to the priority policy is divided into two steps. First, the effectiveness of the resource manager is demonstrated using a pair of symmetric stable workloads with different priorities. In the second step, two different workloads are used, and the high-priority workload is time-varying. This scenario is investigated with limited CPU resources to illustrate the possible performance gains with the work-conserving resource manager. In this section, the workload with a higher priority is referred to as the primary workload, while the lower priority one is referred to as the secondary workload. The experiments in this section uses the TPC-E and TPC-H database workloads introduced before in Section 3.4.

In the experiments, the resource manager employs core partitioning to allocate CPU



(a) Core Allocation with the Partitioning Mechanism



(b) Core Allocation with the Quota Mechanism

Figure 5.4: CPU Allocation between Dynamic Workloads with Weights of 3:1

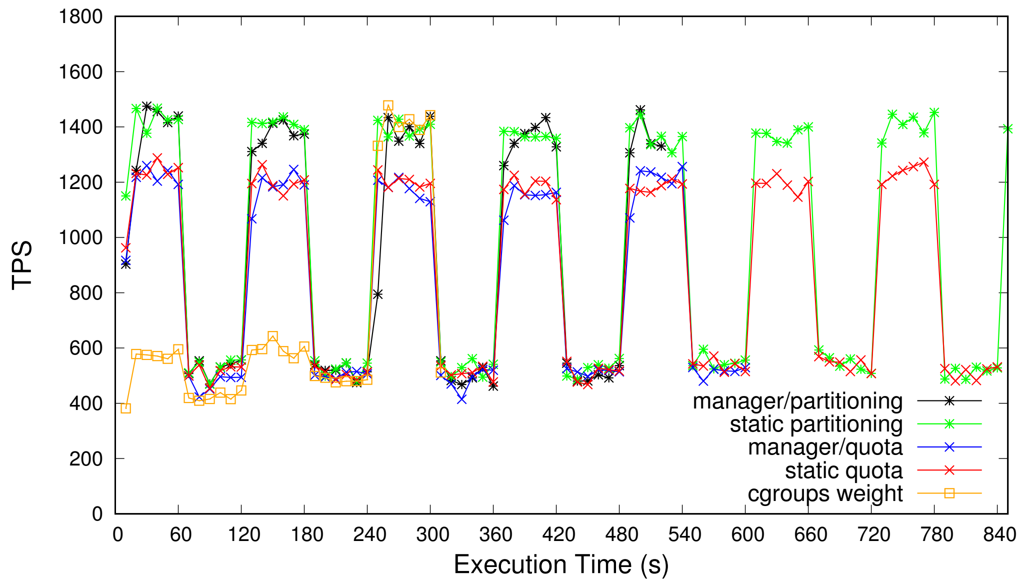


Figure 5.5: Throughput of MariaDB/TPC-E Workload

Table 5.3: Swaptions Completion Time with A Dynamic TPC-E Workload

	manager/partitioning	static partitioning	manager/quota	static quota	cgroups weight
Time	530.08s	859.65s	605.85	847.40s	315.05s

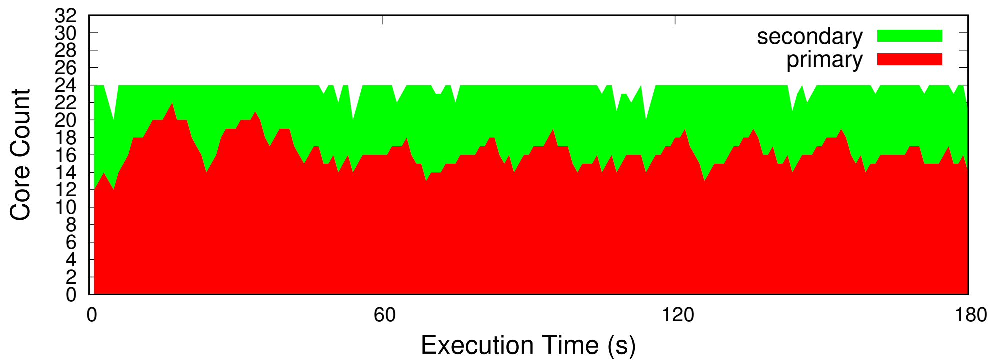
resources. The results with the best static partitioning are examined as a comparison. With the same CPU resources, if the quota mechanism is used for allocation, the primary TPC-E workload requires all the resources to achieve its best performance (as shown in Figure 2.1 in Section 2.2.1). Hence the quota mechanism is not suitable to illustrate how the resource manager works in the two experiments. Different from the proportional sharing experiments, the two mechanisms are not compared in the same experiments.

In current CPU schedulers and resource managers, priorities are usually emulated via weighted time-based allocation [26, 57, 67]. Google's open-source Kubernetes system [26] is a typical example of emulated priority. Therefore, in the two experiments, the results with the resource manager are also compared with the results with a Kubernetes setup. Through different Kubernetes configurations, the primary workload is assigned to the 'Burstable' priority class, while the secondary workload belongs to the 'BestEffort' class. It can be observed that Kubernetes creates sub-groups for the containers in cpuset and cpu subsystem of Linux cgroups. The cpu.shares of the sub-groups in the 'Burstable' class is 10240, while the value of cpu.shares of the sub-groups in the 'BestEffort' class is 2. Meanwhile, any configured limit of CPU resources is implemented by the quota limit. Kubernetes runs applications in Docker containers. For the sake of fairness, the workloads are also executed in Docker containers in the experiments with static partitioning and the resource manager. In the experiments for the other three policies, the applications do not run in Docker containers. In those experiments, it does not matter whether the applications run in Docker containers.

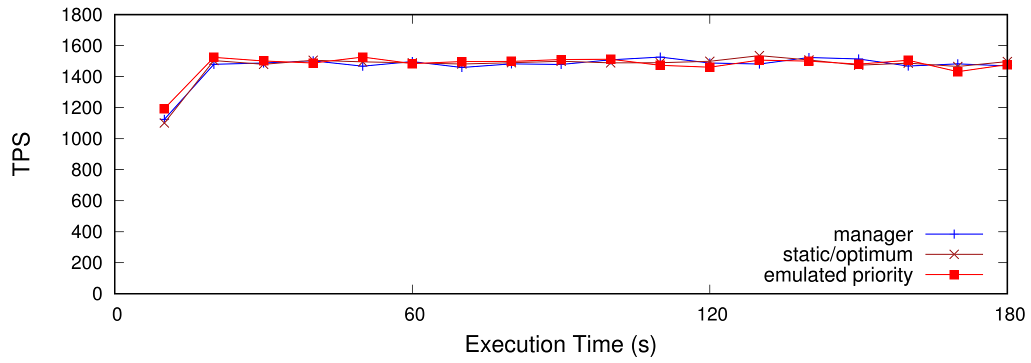
Stable Workloads

The first experiment is used to illustrate that the resource manager can find the proper CPU allocation for two stable workloads based on their priorities. The stable TPC-E workload described in the previous section is used in this experiment. In a preparatory experiment, the throughput of two TPC-E workloads with all the possible static partitioning combinations is measured. When running in parallel with a secondary workload, the throughput of the primary TPC-E workload reaches its peak value when 16 cores are allocated. The two database servers share 24 cores in total in this experiment, which ideally leaves 8 cores for the secondary workload.

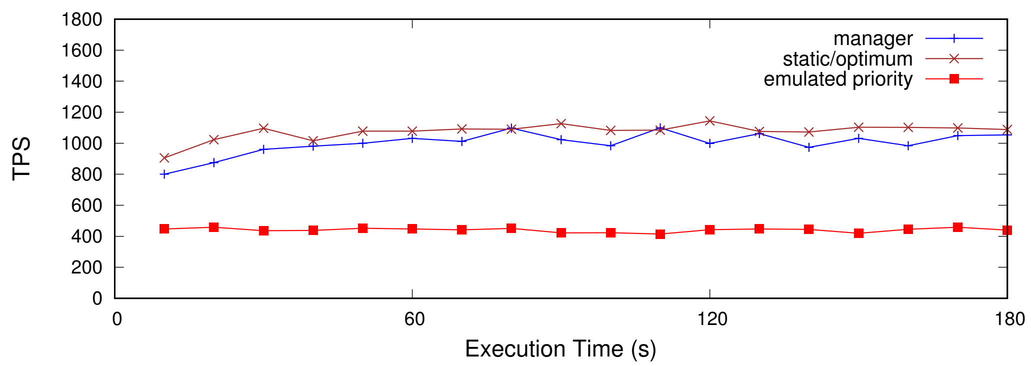
Figure 5.6(a) shows the detailed CPU core allocation during the experiment. At the beginning, the primary workload and the secondary workload are both allocated 12 cores. The resource manager quickly converges to the proper partitioning and keeps the allocation around that point. The spikes in the figure are trial allocations from the multi-timescale mechanism in the tenant controllers. The average number of cores allocated by the resource



(a) Core Allocation



(b) Primary MariaDB/TPC-E Throughput



(c) Secondary MariaDB/TPC-E Throughput

Figure 5.6: Results for Stable Workloads with Different Priorities

manager to the primary workload is 16.2, which is close to the best static allocation. The experiment with the resource manager is repeated 20 times to study the repeatability of the results. The coefficient of variation of the average core allocation is 0.02 for the primary workload, and 0.05 for the secondary workload.

The throughput results of the primary TPC-E workload with different allocation methods are presented in Figure 5.6(b), while the results for the secondary workload are shown in Figure 5.6(c). Each experiment run lasts for 180 seconds. It can be seen that for both the primary and secondary workloads, the resource manager performs almost the same as the best static partitioning. With the resource manager, the average throughput of the primary workload is 98.6% of the result with the best static partitioning. Emulated priority in Kubernetes also performs very well for the primary workload. As for the secondary workload, the average throughput with the resource manager is 94.5% of the result with the best static allocation. However, with emulated priority, the throughput of the secondary workload is much lower, only about 40.1% of the throughput with the best static partitioning (also shown in Figure 5.6(c)). As discussed in Section 2.2.1, when too many CPU cores are allocated to the TPC-E workload, the excessive parallelism causes CPU cycles to be wasted. Dynamic partitioning in the resource manager can effectively limit the parallelism. By comparison, with the emulated priority (i.e., weighted time-based allocation), the primary workload can use all the available cores in parallel. Therefore, a considerable number of CPU cycles are wasted, which are then not available for the secondary workload. These results demonstrate that the resource manager can find the proper core allocation to guarantee the performance of the primary workload, while achieving a better performance of the secondary workload compared to the emulated priority in Kubernetes.

Dynamic Workloads

This experiment creates a setup comprised of OLTP and OLAP workloads that mimics a practical scenario where a multi-tenancy system handles front-end business transactions as well as back-end analytical queries. The performance of the transactional workload must be guaranteed, while the analytical queries are also expected to finish as soon as possible. In this experiment, the primary TPC-E workload has a higher priority and time-varying requirements. The primary dynamic TPC-E workload is the same as in the proportional sharing experiment. This workload has a high-demand phase and a low-demand phase alternatively. The secondary OLAP workload is the TPC-H workload introduced in Section 3.4, which is stable and runs until completion.

To show the performance improvement resulting from the work-conserving resource manager, a resource-constrained scenario is created. The total number of cores for both

workloads is set to 22. The best static allocation in the two phases of the TPC-E workload is 16 and 7 cores respectively. The best static allocation for the TPC-H workload is 15 cores. To guarantee the throughput of the primary workload during the high-demand phases, the reference static allocation is 16 cores for the primary TPC-E workload and 6 cores for the secondary TPC-H workload. The emulated priority configuration is set up using Kubernetes, as before, and both tenants can access all the 22 cores.

Figure 5.7 shows how the resource manager allocates cores to the two workloads. The average number of cores allocated to the TPC-E workload is 15.57 in the high-demand phase and 6.77 in the low-demand phase, close to the ideal static allocation. The average number of cores allocated to the TPC-H workload is 9.87, almost 4 more cores than the static partitioning. Therefore, this workload is able to finish more quickly. With the same experiment repeated 20 times, the coefficient of variation of the average number of cores allocated to the TPC-E workload is 0.04.

The TPS of the TPC-E workload is calculated and plotted every 10 seconds. As shown in Figure 5.8, the resource manager can find a proper core allocation within the first 10 seconds. After that, the throughput of the primary workload is as good as the result using static allocation. In the experiment run shown in this figure, the average throughput during the high-demand phases reaches 96.8% of the average throughput with the static allocation. In this scenario, emulated priority in Kubernetes can also guarantee that the primary workload has enough CPU time. The average throughput is also as good as that with static allocation.

Table 5.4 shows the TPC-H completion time with the three different resource allocation methods. In this scenario, the resource manager reduces the completion time of the secondary OLAP workload by 27.1% compared to the static allocation and 45.2% compared to emulated priority (Kubernetes). The results show that both static partitioning and emulated priority limit the performance of the low-priority tenant, while the resource manager allows the secondary workload to access CPU resources not currently being used by the primary workload.

In summary, the resource manager can still achieve both resource isolation and work conservation with the priority policy. Moreover, the experiments in this section shows that the resource manager can make use of partitioning to avoid the impact of excessive parallelism. Conversely, emulated priority suffers from the parallelism issue in the experiments. CPU resources are wasted rather than used for the secondary workload.

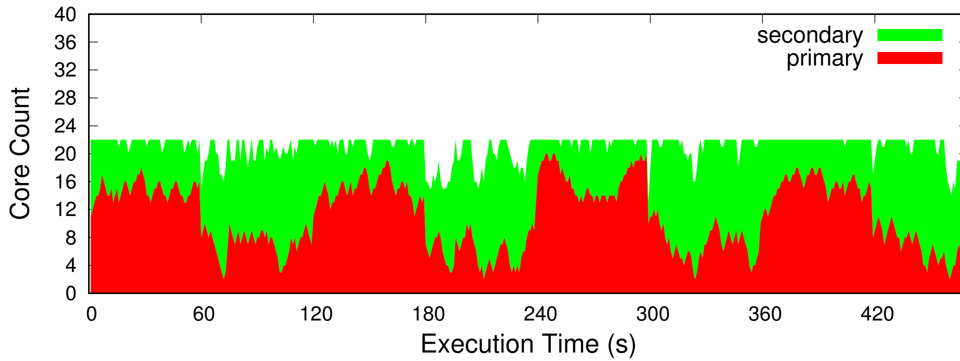


Figure 5.7: Core Allocation with Dynamic Workloads

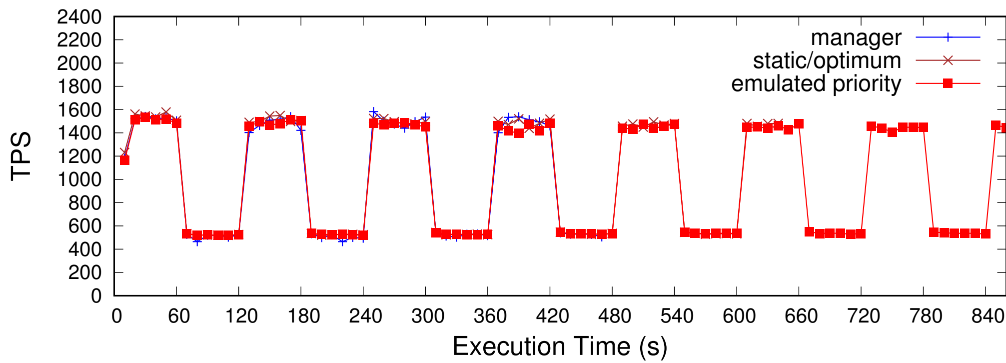


Figure 5.8: Primary MariaDB/TPC-E Throughput

Table 5.4: Secondary MariaDB/TPC-H Completion Time

	resource manager	static partitioning	emulated priority
Time	474.68s	651.57s	866.41s

5.3.4 Guarantee Policy

This section presents the two evaluation experiments for the guarantee policy. In the first experiment, a stable Memcached workload without any rate limit is co-located with the same swaptions benchmark as in Section 5.3.2. The Memcached workload generated by the Mutilate generator [53] is a read-heavy workload with an update ratio of 0.1 (cf. Section 2.2.1). In the second experiment, a dynamic Memcached workload consisting of multiple 120-second cycles is employed. In each cycle, there are a 60 seconds high-demand phase that has no rate limit and a 60 seconds low-demand phase that has a rate limit of 400,000 queries per second.

In the experiments, the resource manager uses the quota mechanism in Linux cgroups to allocate CPU resources. In each experiment, two applications can access 32 physical cores, but the sum of their quota limits is no more than 20 effective cores. The guaranteed quantity for the static or dynamic Memcached workload is set to 12 effective cores, while the swaptions workload is guaranteed to obtain CPU resources equivalent to 8 cores. When the stable Memcached workload is executed in isolation, it requires 20 effective cores. In the low-demand phase, the dynamic Memcached workload requires only 6 effective cores. The swaptions workload requires 26 effective cores when it runs in isolation.

The resource manager is compared with the static quota mechanism in this section. Static quota is widely used in practice to restrict the CPU usage of tenants, but the quota values are fixed. As a comparison, the experiments are also conducted with the weight mechanism in Linux cgroups using a weight ratio of 12:8, because this mechanism is work-conserving and usually used for dynamic workloads.

Stable Workloads

This experiment shows that the resource manager can enforce the guaranteed quantity as well as static quota. The guaranteed CPU allocation for each of the workloads is chosen to be smaller then their respective requirements. According to the guarantee policy, the resource manager should allocate 12 effective cores to the Memcached workload and 8 effective cores to the swaptions workload. As a comparison, the static quota values for the two workloads are directly set to 12 and 8 effective cores.

Figure 5.9 shows the CPU allocation from the resource manager. The initial allocation is that each workload obtains 10 effective cores. The resource manager finds the ideal allocation quickly. The average number of effective cores allocated to the stable Memcached workload is 11.75. The swaptions workload obtains 7.89 effective cores. The resource

manager basically enforces the guaranteed quantity of each workload. The same experiment is repeated 20 times. For the two workloads, the coefficients of variation of CPU allocation are both less than 0.01.

Figure 5.10 illustrates the throughput results of the Memcached workload with three different mechanisms. The result with the resource manager is 96.7% of that with the ideal static quota. In contrast, the weight mechanism cannot guarantee the CPU resources allocated to the Memcached workload. The throughput result with this mechanism is about 25% lower than the result with static quota until the swaptions workload completes. Table 5.5 presents the completion time of the swaptions workload with three different mechanisms. The result with the resource manager is close to that with static quota. With the weight mechanism, the swaptions workload consumes more than 8 effective cores, which completes sooner but influences the performance of the Memcached server.

Dynamic Workloads

This experiment shows how the resource manager ensures the guaranteed CPU resources as well as keeping its work-conserving property. The experiment uses a dynamic Memcached workload and a stable swaptions workload, while the two workloads are still guaranteed to obtain 12 and 8 effective cores respectively. The dynamic Memcached workload requires less than 12 effective cores during the low-demand phase. Hence the swaptions workload can utilize more resources than 8 effective cores during that phase. The prototype uses the proportional sharing policy for the sharing of idle CPU resources from the Memcached workload. The swaptions workload is the only unsatisfied tenant in this experiment, so it can obtain all the extra resources.

Figure 5.11 shows the detailed CPU allocation during the whole experiment. When the workloads both require more CPU resources than their guaranteed quantity, they obtain respective guaranteed quantity of CPU resources. In the high-demand phase, on average there are 11.57 effective cores allocated to the Memcached workload and 7.99 effective cores allocated to the swaptions workload. The swaptions workload only consumes more than 8 effective cores when the requirement of the Memcached workload declines to about 6 effective cores in the low-demand phase. The CPU time allocated to the Memcached workload is not more than 12 effective cores except at the end of the experiment when the requirement of the swaptions workload declines. The same experiment is also repeated 20 times. Regrading the CPU allocation to the two workloads, the coefficient of variation is 0.03 and 0.02 respectively.

Figure 5.12 illustrates the throughput result of the Memcached workload with different

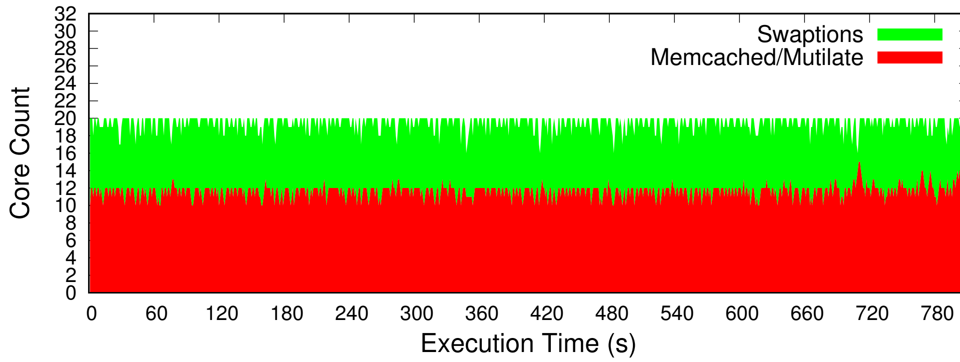


Figure 5.9: Core Allocation between Stable Workloads with Guaranteed CPU Resources

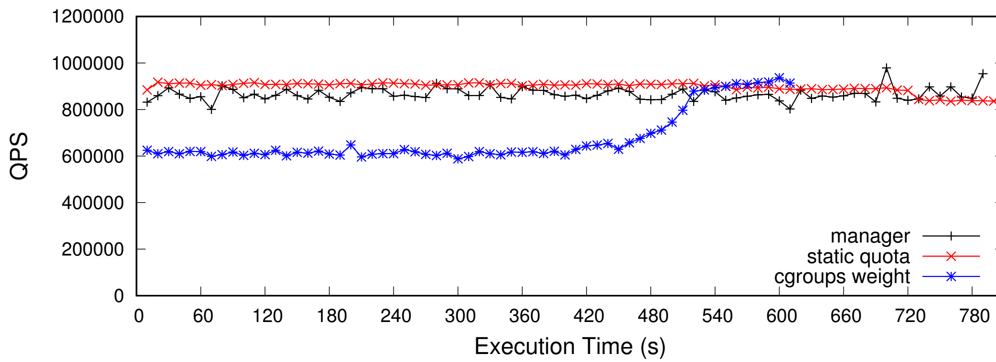


Figure 5.10: Throughput of Memcached/Mutilate Workload

Table 5.5: Swaptions Completion Time with Stable Memcached/Mutilate Workload

	resource manager	static quota	cgroups weight
Time	813.79s	830.97s	641.94s

allocation mechanisms. The average QPS in the high-demand phase with the resource manager is 96.4% of that with static quota. The average QPS with the weight mechanism is only 82.8% of that with static quota. In the low-demand phase, the QPS results with different mechanisms are almost the same. The corresponding completion times of the swaptions workload are presented in Table 5.6. The time result with the resource manager is 23% shorter than the result with static quota because the resource manager is work-conserving. The time result with the weight mechanism is shortest because the guaranteed quantity of the Memcached workload is not enforced, and the swaptions workload consumes extra CPU resources.

5.3.5 Efficiency Policy

This section evaluates how the resource manager performs under the efficiency policy. A total of 20 effective cores are shared by two tenants. The CPU resources are allocated via the quota mechanism. Two scalable workloads are chosen to illustrate the effect of the resource manager with respect to efficiency. Two different server applications MariaDB and Memcached are used in this section. The workload for the MariaDB server is the Sysbench OLTP benchmark [40], which is designed to test MySQL/MariaDB. The workload for the Memcached server is the same read-heavy workload generated by Mutilate [53] in the previous section.

The first experiment aims to test whether the resource manager can find the most efficient allocation between two stable workloads in terms of the total normalized score. Because both the workloads are fairly scalable, the total normalized score will be higher when more CPU resources are allocated to the workload with a larger efficiency factor. Without any rate limit in the workload generators, the most efficient CPU allocation is extreme like 19 cores for one workload and 1 core for the other workload. When a rate limit is introduced to one workload generator, this workload's scalability changes, and thus the distribution of the total normalized score shifts towards the middle. In this experiment, a rate limit of 1,000,000 queries per second is set in the Mutilate workload generator to verify that the resource manager can find the most efficient allocation that is manually set.

The purpose of the second experiment is to show the benefit of adaptive CPU allocation with respect to efficiency. In this experiment, a dynamic Mutilate workload is executed with the stable Sysbench workload above. The dynamic Mutilate workloads consists of two 120-second cycles. There is no rate limit in the first 60 seconds of each cycle, while in the next 60 seconds, the workload generator has a rate limit of 220,000 queries per second. In this experiment, the efficiency factors of the Mutilate workload and the Sysbench workload

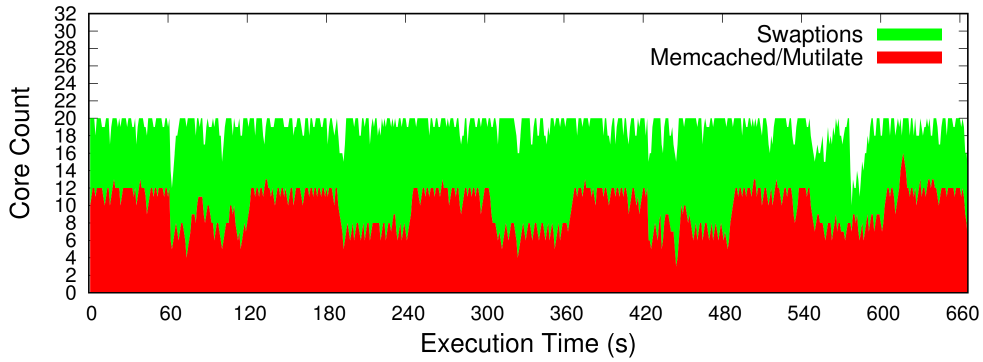


Figure 5.11: Core Allocation between Dynamic Workloads with Guaranteed CPU Resources

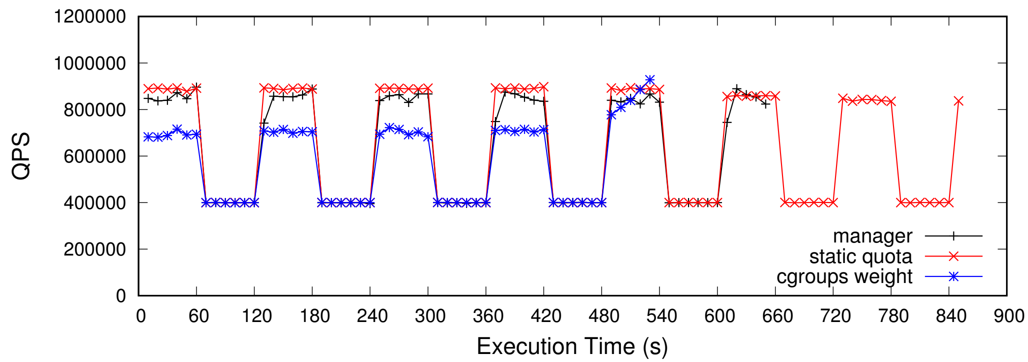


Figure 5.12: Throughput of Memcached/Mutilate Workload

Table 5.6: Swaptions Completion Time with Dynamic Memcached/Mutilate Workload

	resource manager	static quota	cgroups weight
Time	669.71s	872.30s	555.26s

are set to 2 and 1 respectively. As the Mutilate workload changes during the experiment, the most efficient CPU allocation also changes. The work-conserving resource manager employed here can monitor the changes of runtime performance and adjust CPU allocation accordingly.

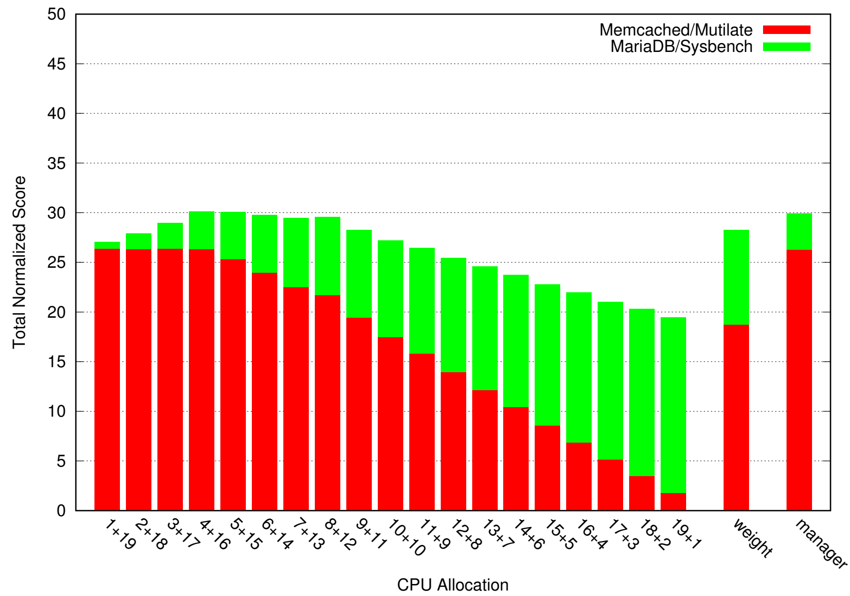
The two experiments are also repeated with the weight mechanism in Linux cgroups. The weight ratio is the same as the ratio of efficiency factors. The results are presented in the following sections as a comparison.

Stable Workloads

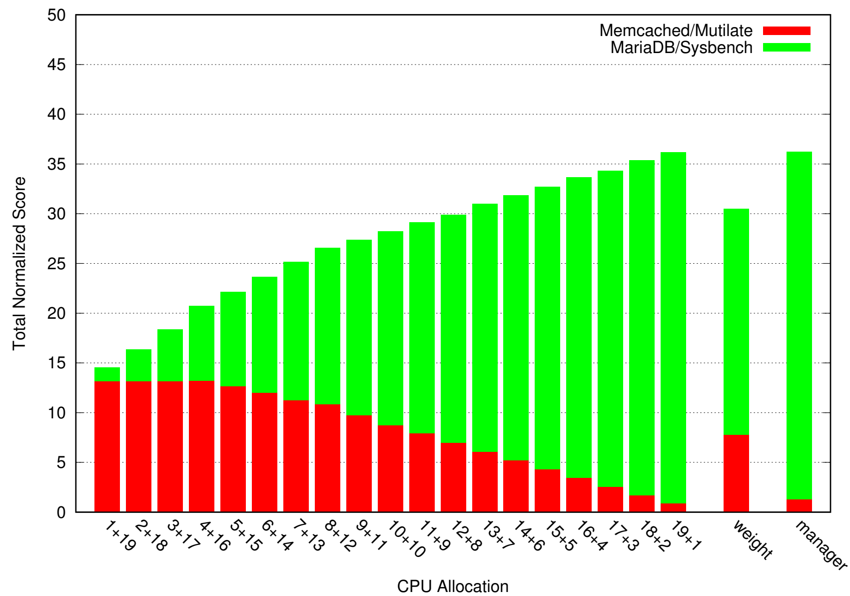
In this experiment, the performance of the two workloads is measured with different static CPU allocations, and then the normalized scores are calculated. In the first group, the efficiency factor of the Sysbench workload is 1, while the efficiency factor of the Mutilate workload is 2. In the second group, the factor setting is reversed. Figure 5.13 shows total normalized scores with different static allocations. A static allocation m+n in this figure means m effective cores are allocated to the Sysbench workload and n effective cores are allocated to the Mutilate workload. Generally speaking, the total normalized score is higher when more CPU resources are allocated to the workload with an efficiency factor of 2. In Figure 5.13(a), because of the rate limit in the Mutilate workload, the best static allocation is 4 effective cores for the Sysbench workload and 16 effective cores for the Mutilate workload. In Figure 5.13(b), the best static allocation is 19 effective cores for the Sysbench workload and 1 effective core for the Mutilate workload. The resource manager can find the proper allocation in both cases. By comparison, with the weight mechanism, every workload gets about half of the CPU resources regardless of its efficiency factor. Thereby the total normalized score is lower than the result with the resource manager. The weight mechanism is not suitable for the implementation of the efficiency policy. The resource manager performs consistently with these stable workloads. With a set of 20 samples, the coefficient of variation of the total normalized score is less than 0.01 when the factor ratio is either 1:2 or 2:1.

Dynamic Workloads

The second experiment verifies that when the most efficient allocation dynamically changes, the resource manager can adjust its CPU allocation accordingly. In this experiment, the efficiency factor of the Mutilate workload is 2. In the high-demand phase of the Mutilate workload, it is more efficient to allocate CPU resources to the Mutilate workload. In



(a) Efficiency Factors of Sysbench and Mutilate Workloads are 1:2

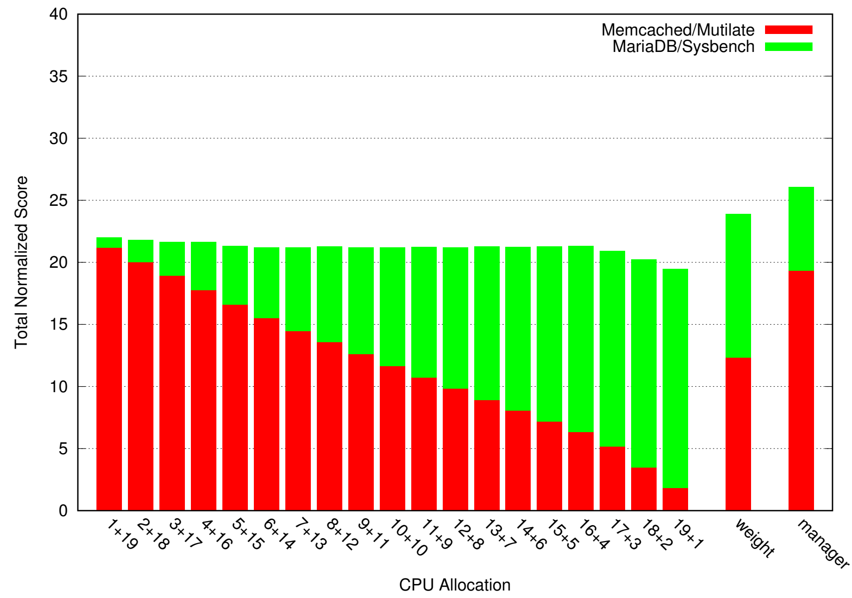


(b) Efficiency Factors of Sysbench and Mutilate Workloads are 2:1

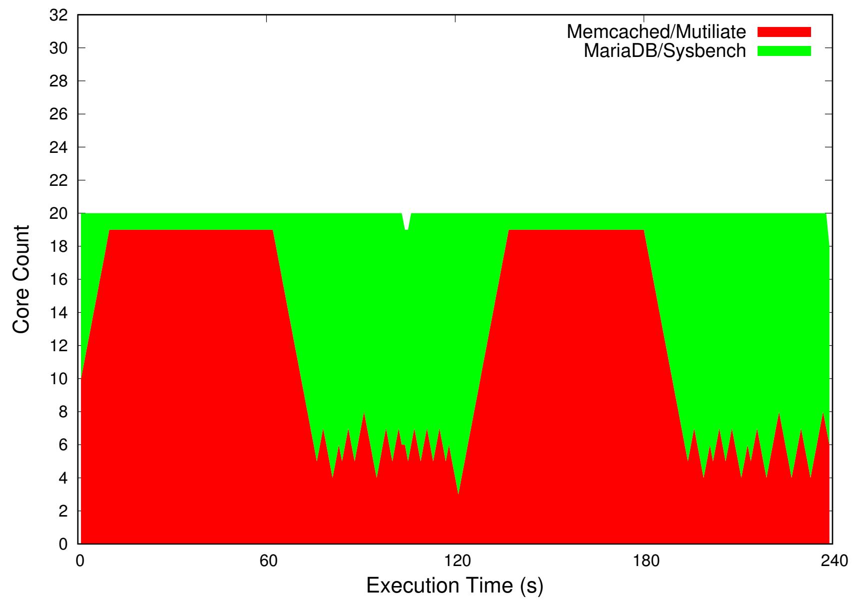
Figure 5.13: Efficiency with Stable Workloads

the low-demand phase, however, the most efficient allocation is 4 effective cores for the Mutilate workload and 16 effective cores for the Sysbench workload. No static allocation is suitable for this kind of dynamic workload. The total normalized scores shown in Figure 5.14(a) confirm that the resource manager achieves a much better result based on online performance measurement. The total normalized score is 18.3% higher than the best result with static allocations. The weight mechanism is also work-conserving, so the Sysbench workload can obtain more CPU resources. The score with the weight mechanism is higher than the best result with static allocations, but lower than the result with the resource manager.

Figure 5.14(b) shows the CPU allocation in different phases. The efficiency-based allocation algorithm supports adaptive CPU resource allocation. When the requirement of the Mutilate workload decreases, the algorithm successfully finds the new efficient CPU allocation. The algorithm still performs consistently with dynamic workloads. With the same experiment repeated 20 times, the coefficient of variation of the total normalized score is only 0.01.



(a) Efficiency Factors of Sysbench and Mutilate Workloads are 1:2



(b) CPU Allocation

Figure 5.14: Efficiency with Dynamic Workloads

Chapter 6

I/O Bandwidth Management

The previous chapters in this thesis study how to achieve both resource isolation and work conservation with respect to CPU resource management. CPU is not the only resource required by applications, and other types of resources may also become a bottleneck. When multiple applications share resources, resource isolation and work conservation are also desirable in the management of non-CPU resources.

This chapter studies whether the same feedback-based methodology introduced above is useful for the management of other types of resources. UIPT can be used to estimate application performance regardless of the resource type, as long as there is a good correlation between UIPT and the application-level metric. The adaptive allocation algorithms may need to be tuned with respect to concrete resources. In addition, the methodology in this thesis is based on an assumption that there is an existing mechanism to effectively restrict the resources available to a workload.

The management of I/O bandwidth is studied as an example in this chapter. Nowadays an enormous volume of data is generated and used in real world applications. Data-intensive applications compete fiercely for access to large volumes of data on secondary storage such as hard disk drives (HDD) and solid state disks (SSD). I/O contention has a significant impact on application performance [39]. I/O bandwidth management must address problems similar to those of CPU management. This preliminary study shows that the methodology presented in this thesis is a promising approach for a better management of I/O bandwidth.

While it is possible to design a resource manager that manages different types of resources, currently the I/O bandwidth manager in this chapter is not integrated with the CPU manager. UIPT provides a feedback independent of resource types, and facilitates

a uniform interface to manage different types of resources. Yet the adaptive allocation algorithms proposed in this thesis only search along one dimension. There are two potential solutions to this limitation. First, in one stage, an application usually has only one bottleneck resource. For example, CPU is not a constraint when an I/O-intensive application spends plenty of time waiting for I/O operations to finish. Therefore, the adaptive allocation algorithms can be used to manage different types of resources at different times. Alternatively, a multi-dimensional search algorithm is another possible solution. The search space of the resource allocation problem is not very large, and sampling techniques may help reduce the search time. Future research needs to be carried out to manage multiple types of resources based on feedback.

6.1 I/O Bandwidth Control

To manage I/O requests to diverse underlying devices, operating systems employ a generic layer between file systems (or other libraries) and the specific device drivers. This layer provides a level of abstraction for the hardware responsible for retrieving and storing data. Block devices are a common abstraction used in the I/O management layer. In block I/O, data are buffered first and then an entire block of data are read or written. This research focuses on Linux block I/O because this system is mature and widely used.

The Linux block layer provides a uniform way for applications to access storage devices. I/O requests from applications are buffered in a request queue, and device drivers receive requests from the head of the queue. While requests are in the queue, the block layer performs I/O scheduling to improve the overall I/O performance. For instance, requests can be sorted to minimize seek operations; requests for adjacent sectors can be merged; fairness policies and bandwidth limits can be applied. This single-queue block layer has several different I/O schedulers. The Complete Fairness Queuing (CFQ) scheduler is the default I/O scheduler [8]. This scheduler separates I/O requests according to their processes and allocates time slices to each process based on their weights. The single-queue block layer was designed many years ago and optimized for HDD performance. With the development of high speed modern devices with internal data parallelism, a new generation multi-queue block layer with a submission queue per core is introduced into Linux. The I/O schedulers are also redesigned for the multi-queue layer.

The main objective of I/O schedulers is to optimize the overall performance of block I/O. When multiple I/O-intensive applications run simultaneously, their I/O requests compete for the request queue in the block layer and impact the performance of each other. Current I/O schedulers do not provide effective resource isolation. Proportional sharing is

only supported in the CFQ scheduler. CFQ assigns dedicated time slices to processes based on their weights. However, this scheduler is not work-conserving. It does not dynamically adjust the time slice according to the level of disk contention or the characteristics of requests. Even when a task has an underload, the time slice is reserved for the task, and thus the I/O bandwidth is wasted.

It is challenging to tune I/O scheduling algorithms for accurate bandwidth sharing among tasks. The *blkio* subsystem of Linux cgroups has a weight mechanism for proportional I/O bandwidth sharing. This mechanism only works when the CFQ I/O scheduler is used in the block layer, and may cause waste of I/O bandwidth because CFQ is based on time reservation. Similar to CPU management, I/O bandwidth isolation relies on throttling in practice. I/O bandwidth throttling in the blkio subsystem can set an upper limit on the read and write operations that a task can perform. The limit is specified in operations per second or bytes per second. This I/O throttling mechanism is similar to the quota mechanism in the cpu subsystem of Linux cgroups. A constant limit is not work-conserving, and may result in poor I/O performance due to the changing resource requirements of applications.

The feedback-based methodology presented in this thesis is suitable for the control of I/O bandwidth. The limit of I/O bandwidth for an application can be dynamically adjusted according to its requirement. UIPT can be used to estimate the application performance regardless of the type of bottleneck resource. For an I/O-intensive application, when I/O requests cannot be finished quickly, the execution speed of the application will decrease, and UIPT will decrease as well. In contrast, an application-specific performance metric is used in a previous study to reduce the impact of I/O contention by automatically tuning the thread pool in Spark executors [39]. Such a method is specific to Spark and does not work for a generic resource manager.

The algorithms introduced in Chapter 4 can also be used to adaptively adjust I/O bandwidth limits. Excessive CPU resources may increase application parallelism, i.e., the number of threads running in parallel, and aggravate contention on shared resources, and thus cause a decline in performance. Different from CPU resources, I/O bandwidth does not directly affect application parallelism. When the bandwidth limit is greater than the application requirement, the application performance does not decline. Thereby, the performance curve with I/O bandwidth as the X axis first shows an increase and then becomes flat. The adaptive allocation algorithms are able to handle this kind of curve with a plateau (cf. Chapter 4).

A prototype for I/O bandwidth control is developed based on the hill-climbing algorithm described in Section 4.2. The I/O bandwidth allocation is done via the blkio subsystem of

Linux cgroups. The prototype employs the throttling mechanism. However, the prototype here only shows the effectiveness of the methodology in the block layer, because the blkio subsystem does not manage the OS buffer cache. I/O contention in the OS buffer cache significantly affects the performance of I/O-intensive applications, but there is no existing mechanism that can effectively restrict I/O operations in both the block layer and the buffer cache.

6.2 Experimental Evaluation

This section evaluates whether the UIPT metric and the hill-climbing algorithm work for an I/O-intensive application. The system employed in the experiments is a persistent key-value store called *etcd* (version 3.3.10) [21]. In etcd, the *fsync* system call is invoked frequently, especially as part of its implementation of Raft protocol. According to the documentation [22], etcd is very sensitive to the performance of disk writes. In the experiments, the I/O throttling mechanism is used to control the bandwidth available to the etcd server, and the rate of the write operations is specified in bytes per second. The workload employed in the experiments is the "put" workload generated using the benchmark tool included with etcd. The generator uses 256 connections and inserts a total of 3,000,000 records. The key size and the value size in each record are both 256 bytes. The machine and Linux system introduced in previous chapters are used for experiments in this section. One processor with 16 cores and 2 NUMA nodes are allocated to the etcd server exclusively. CPU and memory resources are sufficient for this workload, and the bottleneck is I/O bandwidth.

6.2.1 UIPT

The application-level performance metric of the etcd put workload is requests per second (RPS). This experiment evaluates the correlation between UIPT and RPS. In the experiments, the bandwidth limit of write operations is varied from 2 to 50 Mbps. UIPT is measured every second while the etcd server is processing the workload. RPS is the average result of the whole process. The average RPS and average UIPT results with different bandwidth limits are illustrated in Figure 6.1. The bars show 95% confidence interval of UIPT results. The two metrics clearly show the same trend in the figure. The correlation coefficient between the average UIPT and the average RPS is 0.99. More importantly, the relative change in each metric when 2 Mbps are added to the bandwidth limit is calculated. The correlation coefficient between the relative changes in average UIPT and the relative

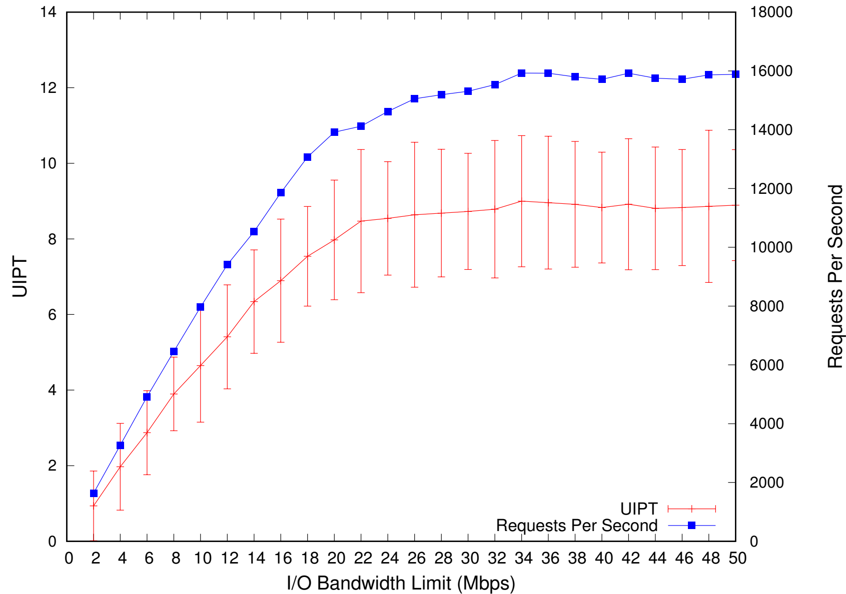


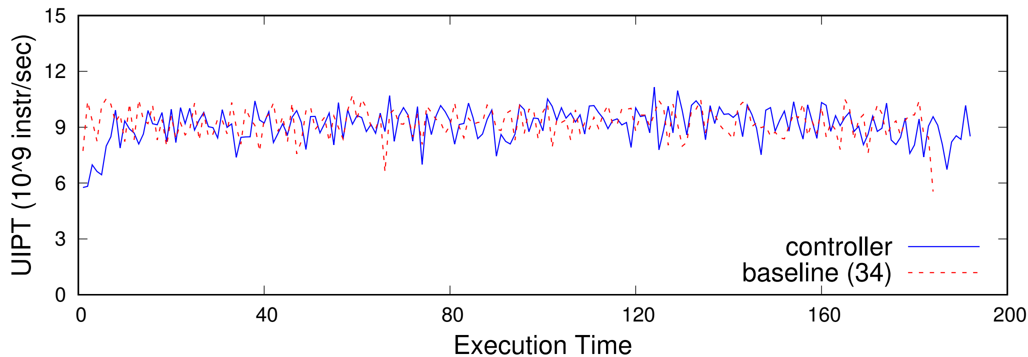
Figure 6.1: UIPT for Etcd Put Workload

changes in average RPS is also 0.99. There exists a strong correlation between the two metrics. UIPT is a good proxy of RPS for this kind of I/O-intensive workload. This figure also confirms that the performance curve contains a plateau area. Moreover, even with a static write bandwidth limit, the variation of the UIPT change is large, which increases the difficulty of the adaptive allocation problem.

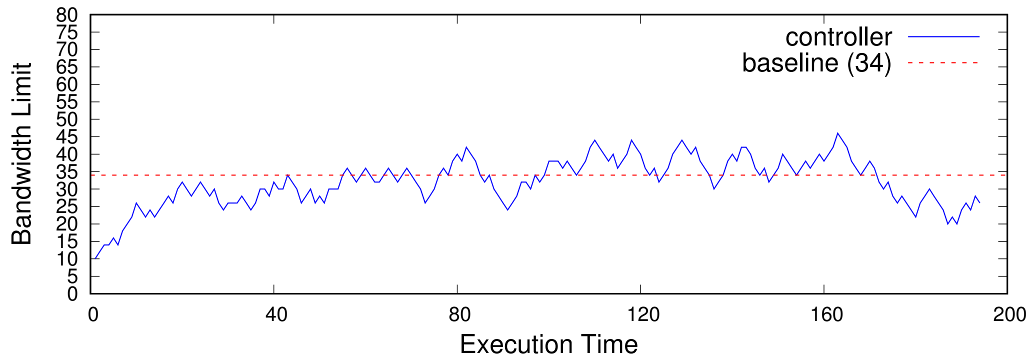
6.2.2 Adaptive I/O Bandwidth Throttling

This experiment examines whether the controller based on the hill-climbing algorithm can find the proper limit of write operations. The same etcd put workload used in the previous experiment is used here. The ideal (smallest) bandwidth limit to achieve the highest throughput is 34 Mbps. The controller changes the write bandwidth limit by 2 Mbps in each interval. The experiment is repeated 20 times respectively with two different initial bandwidth limits. One is lower than the ideal static allocation (10 Mbps), while the other one is higher (60 Mbps).

The application performance with the controller is close to the baseline performance at both the application level and the instruction level. The average throughput of 40 runs reaches 97.5% of the baseline performance. The coefficient of variation is only 0.02, so the



(a) UIPT of the Etcd Workload

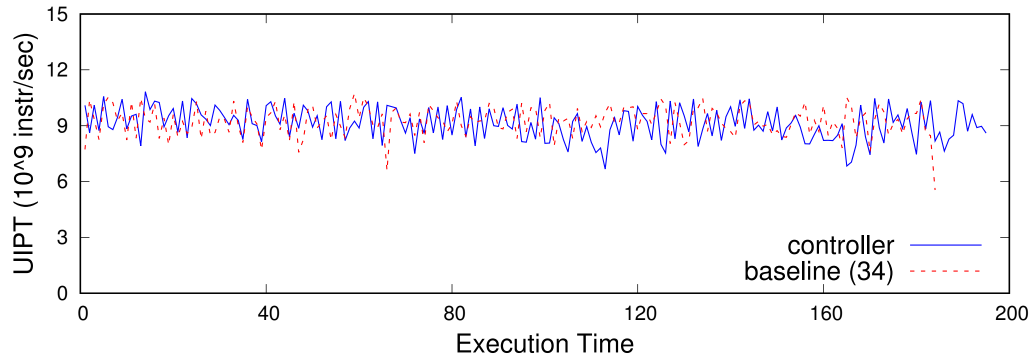


(b) Bandwidth Limits

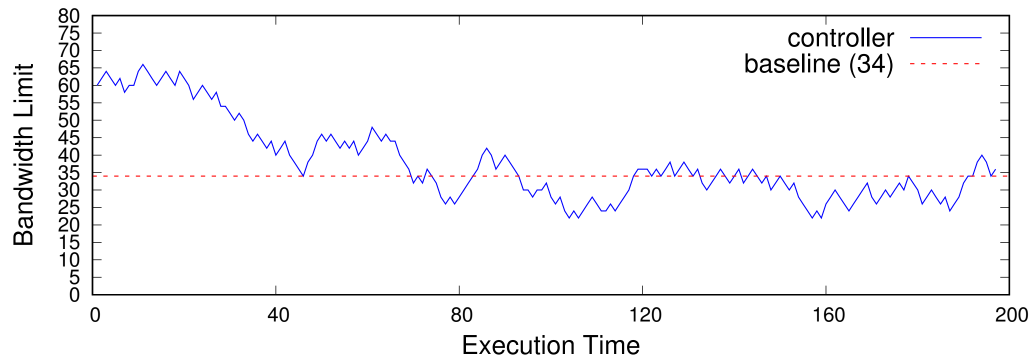
Figure 6.2: Etcd Workload: Start from 10 Mbps

controller consistently guarantees the throughput of the etcd workload. The measurements of UIPT during two experiment runs are depicted in Figure 6.2(a) and Figure 6.3(a). The baseline UIPT with a limit of 34 Mbps is also plotted in the figures as a reference. Although the UIPT of this workload has highly frequent fluctuations even with a fixed limit of write bandwidth, the controller finds the ideal bandwidth limit and effectively keeps the throughput of the workload within a reasonable range.

Figure 6.2(b) illustrates the detailed bandwidth limits during the experiment with an initial allocation of 10 Mbps. Without any prior knowledge about the ideal bandwidth limit, the controller can converge toward the ideal allocation. The time for the controller to reach the ideal allocation depends on the initial limit. In the experiments starting from



(a) UIPT of the Etcd Workload



(b) Bandwidth Limits

Figure 6.3: Etcd Workload: Start from 60 Mbps

10 Mbps, the search process takes 22 intervals on average. In contrast, as shown in Figure 6.3(b), it takes longer for the controller to search in the other direction. When starting from 60 Mbps, the controller takes 46 intervals on average to find the ideal bandwidth limit. The highly frequent fluctuations of UIPT may interfere with the hill-climbing algorithm. The algorithm cannot distinguish whether the change in UIPT is caused by the change in bandwidth limit or the change in workload.

The average write bandwidth limit after convergence in each run is considered as a sample mean. In the experiments with an initial allocation of 10 Mbps, the mean of 20 sample means is 32.08 Mbps, and the coefficient of variation is 0.05. When starting from 60 Mbps, the mean of 20 sample means is 34.12 Mbps, and the coefficient of variation is 0.07. The controller does not perform as stably as in the CPU resource management, but still achieves a satisfactory result. As shown in Figure 6.1, when the bandwidth limit increases from 22 Mbps to 34 Mbps, the measured UIPT increases on average but the extent of increase is insignificant. The controller lingers in this sub-optimal range. Aside from the fluctuations of UIPT, this is another reason why the controller is not as stable as for the CPU resource management.

Chapter 7

Conclusion

7.1 Summary

The goal of this thesis is to find a general methodology for CPU sharing among multiple tenants, which enforces resource isolation and keeps the work-conserving property as well. This research solves a primary concern of workload consolidation. On the one hand, the effective resource isolation can protect workloads from being affected by each other. On the other hand, resource waste due to overprovisioning can be avoided. An experimental study in this thesis confirms that it is challenging for current general-purpose CPU management systems to achieve the two objectives simultaneously.

Previous research has typically proposed systems that require integration of specific applications. Some proposals build performance models for specific applications or workloads [36, 43, 58, 76]. Some designs need the target performance of specific applications as input, and measure application-level metrics by modifying the applications [34, 41, 58]. The resource manager in this thesis is not specific to any application. This feedback-based system can support various workloads via a uniform application-agnostic performance metric. The UIPT metric can be measured without modifications in the OS kernel or applications. The measurement can provide meaningful information in a short interval to make up for the shortcoming of lagging reaction in feedback-based systems.

This thesis does not rely on any complex system model, but proposes two simple feedback-based solutions to the dynamic optimization problem of adaptive resource allocation. The two algorithms proposed can help detect the CPU requirements that enable applications to achieve their best performance. Because the algorithms do not require any

performance target of specific applications, the resource manager can be used for general-purpose CPU sharing. This thesis presents how the resource manager supports three common sharing policies in practice and a novel policy for general system efficiency. Experiments with representative real-world workloads show the effectiveness of the resource manager in these different scenarios.

7.2 Future Work

First, from the allocation algorithm experiments, the most difficult issue to handle is highly frequent workload changes with a moderate magnitude. When the magnitude of measured UIPT change caused by a workload change is close to that caused by a change in CPU allocation, it is difficult for the feedback-based algorithms to distinguish whether the UIPT change is caused by a change in CPU allocation. If such spurious UIPT changes are measured successively, they will mislead the feedback-based algorithms and slow down the convergence. Although the multi-interval adjustment proposed in this thesis is effective, there is still room for improvement in this aspect. For example, if the multi-interval window in Algorithm 2 is short, the algorithm is not very stable even when the workload is stable. However, if this window is long, the convergence becomes slow. There may be other algorithms to filter out the disturbance from the highly frequent dynamics, for example time series algorithms [58, 66]. The resource manager can be improved if a better adaptive allocation algorithm is designed in the future.

Second, the experimental study in Section 2.2.1 shows the impact of excessive parallelism. With core partitioning, the resource manager can effectively restrict the parallelism. With the quota mechanism, conversely, the resource manager does not control the parallelism directly. However, the quota mechanism is a more generic allocation method since the allocation unit can be a fraction. Only with the quota mechanism, the resource manager can support an arbitrary weight ratio in the proportional sharing scenario. How to combine the two allocation methods is an interesting question. Ideally, the resource manager uses core partitioning to avoid excessive parallelism and uses the quota mechanism to implement the fractional part of CPU allocation.

The third direction is to connect the generic UIPT metric to application-level system goals and make use of the feedback of UIPT changes to fulfill overall system goals. The overall system goal can simply be a specific performance target. It is a useful feature in reality to keep the performance of applications at a target level. Given the UIPT value corresponding to the target application performance, it is straightforward to modify the allocation algorithms, especially the fuzzy controller to regulate the CPU allocation to

meet a performance target of applications. The target UIPT value can be measured either online or offline. The overall system goal can also be the maximum utility or revenue. The relationship between the application performance and the overall utility/revenue can be determined by a S-shaped utility function or a piecewise SLA revenue function. A research direction is enhancing the resource manager to support CPU sharing based on various utility/SLA functions. Section 5.2.4 introduces a sharing policy which uses UIPT changes to estimate the overall system efficiency. In this efficiency policy, there is an assumption that the system utility has a linear relationship with the application performance (cf. Equation 5.1). It requires more effort to convert application-level utility functions or SLA functions to a low-level function of UIPT, so that the resource manager can support a sharing policy based on complex performance trade-offs.

Moreover, as shown in Chapter 6, the methodology in this thesis can be used to manage other types of resources. Yet the prerequisite is that there is an effective and flexible interface to limit the usage of that resource. For example, there is no flexible mechanism in the current Linux system to limit memory usage. Some mechanisms, like the memory subsystem of Linux cgroups, kill a task immediately if the task uses more memory than a defined limit. This kind of mechanism cannot be used for feedback-based control. Further, UIPT can provide a uniform feedback for a resource manager to potentially manage multiple types of resources simultaneously. Such a resource manager requires an algorithm to automatically detect the type of bottleneck resources or search the best resource allocation in a multi-dimensional space.

References

[1] Alaa R Alameldeen and David A Wood. IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro*, 26(4):8–17, 2006.

[2] Cosimo Anglano, Massimo Canonico, and Marco Guazzone. FC2Q: Exploiting Fuzzy Control in Server Consolidation for Cloud Applications with SLA Constraints. *Concurrency and Computation: Practice and Experience*, 27(17):4491–4514, 2015.

[3] Cosimo Anglano, Massimo Canonico, and Marco Guazzone. FCMS: A Fuzzy Controller for CPU and Memory Consolidation Under SLA Constraints. *Concurrency and Computation: Practice and Experience*, 29(5):e3968, 2017.

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-scale Key-Value Store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[5] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. The Datacenter as a Computer: Designing Warehouse-Scale Machines. *Synthesis Lectures on Computer Architecture*, 13(3):1–189, 2018.

[6] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[7] Marcus Carvalho, Walfredo Cirne, Francisco Brasileiro, and John Wilkes. Long-term SLOs for Reclaimed Cloud Computing Resources. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 20:1–20:13, 2014.

[8] Complete Fairness Queueing, 2019. https://www.kernel.org/doc/Documentation/block /cfq-iosched.txt (accessed September 1, 2019).

[9] Linux Control Groups, 2019. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt (accessed September 1, 2019).

[10] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. Comparison of the three CPU schedulers in Xen. *SIGMETRICS Performance Evaluation Review*, 35(2):42–51, 2007.

[11] Linux Containers. LXC Project, 2019. http://linuxcontainers.org (accessed September 1, 2019).

[12] Transaction Processing Performance Council. TPC-E Benchmark, 2019. http://www.tpc.org/tpce/ (accessed September 1, 2019).

[13] Transaction Processing Performance Council. TPC-H Benchmark, 2019. http://www.tpc.org/tpch/ (accessed September 1, 2019).

[14] Carlos Cruz, Juan R González, and David A Pelta. Optimization in Dynamic Environments: A Survey on Problems, Methods and Measures. *Soft Computing*, 15(7):1427–1448, 2011.

[15] Carlo Curino, Evan PC Jones, Samuel Madden, and Hari Balakrishnan. Workload-Aware Database Monitoring and Consolidation. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 313–324. ACM, 2011.

[16] Sudipto Das, Vivek R. Narasayya, Feng Li, and Manoj Syamala. CPU Sharing Techniques for Performance Isolation in Multi-tenant Relational Database-as-a-service. *Proc. VLDB Endow.*, 7(1):37–48, September 2013.

[17] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource- efficient and QoS-aware Cluster Management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, 2014.

[18] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *ACM SIGCOMM Computer Communication Review*, volume 19, pages 1–12. ACM, 1989.

[19] Docker, 2019. http://www.docker.com (accessed September 1, 2019).

[20] MariaDB Docker Official Images, 2019. https://hub.docker.com/_/mariadb (accessed September 1, 2019).

[21] etcd, 2019. https://etcd.io/ (accessed September. 1, 2019).

[22] Hardware Recommendations for ETCD, 2019. https://github.com/etcd-io/etcd/blob/master/Documentation/op-guide/hardware.md (accessed September. 1, 2019).

[23] The MariaDB Foundation. MariaDB, 2019. https://mariadb.org (accessed September 1, 2019).

[24] Free Software Foundation. *TOP(1) Linux User's Manual*. http://man7.org /linux/man-pages/man1/top.1.html (accessed September 1, 2019).

[25] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In *2010 International Conference on Network and Service Management*, pages 9–16. Ieee, 2010.

[26] Google. Kubernetes System, 2017. https://kubernetes.io/ (accessed September 1, 2019).

[27] Grid Engine Documentation, 2019. https://arc.liv.ac.uk/SGE/htmlman/manuals.html (accessed September 1, 2019).

[28] Cong Guo and Martin Karsten. Towards Adaptive Resource Allocation for Database Workloads. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2015*, pages 49–60, 2015.

[29] Red Hat. libvirt: the Virtualization API, 2019. https://libvirt.org/ (accessed September 1, 2019).

[30] Joseph L Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[31] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 22:1–22:14, 2011.

[32] Heath-Jarrow-Morton Framework, 2019. https://en.wikipedia.org/wiki/Heath-Jarrow-Morton_framework (accessed September. 1, 2019).

[33] Henry Hoffmann, Jonathan Eastep, Marco D Santambrogio, Jason E Miller, and Anant Agarwal. Application Heartbeats: A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments. In *Proceedings of the 7th international conference on Autonomic computing*, pages 79–88. ACM, 2010.

[34] Henry Hoffmann, Martina Maggio, Marco D Santambrogio, Alberto Leva, and Anant Agarwal. SEEC: A Framework for Self-aware Management of Multicore Resources. *MIT CSAIL Technical Report*, 2011.

[35] David Jackson, Quinn Snell, and Mark Clement. Core Algorithms of the Maui scheduler. In *Job Scheduling Strategies for Parallel Processing*, pages 87–102. Springer, 2001.

[36] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. Adaptive Resource Provisioning for Virtualized Servers Using Kalman Filters. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 9(2):10:1–10:35, 2014.

[37] Martin Karsten. Approximation of Generalized Processor Sharing with Interleaved Stratified Timer Wheels. *IEEE/ACM Transactions on Networking (TON)*, 18(3):708–721, 2010.

[38] Srinivasan Keshav. *An Engineering Approach to Computer Networking*, volume 1. Addison-Wesley Reading, 1997.

[39] Sobhan Omranian Khorasani, Jan S Rellermeyer, and Dick Epema. Self-adaptive Executors for Big Data Processing. In *Proceedings of the 20th International Middleware Conference*, pages 176–188, 2019.

[40] Alexey Kopytov. Sysbench Manual. *MySQL AB*, 2012.

[41] Mustafa Korkmaz, Martin Karsten, Kenneth Salem, and Semih Salihoglu. Workload-Aware CPU Performance Scaling for Transactional Database Systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 291–306. ACM, 2018.

[42] Linux Kernel Virtual Machine, 2019. https://www.linux-kvm.org (accessed September 1, 2019).

[43] Palden Lama and Xiaobo Zhou. Aroma: Automated Resource Allocation and Configuration of Mapreduce Environment in the Cloud. In *Proceedings of the 9th international conference on Autonomic computing*, pages 63–72. ACM, 2012.

[44] Jacob Leverich and Christos Kozyrakis. Reconciling High Server Utilization and Submillisecond Quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 4:1–4:14, 2014.

[45] Min Li, Liangzhao Zeng, Shicong Meng, Jian Tan, Li Zhang, Ali R Butt, and Nicholas Fuller. MRONLINE: MapReduce Online Performance Tuning. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 165–176. ACM, 2014.

[46] Tong Li, Dan Baumberger, and Scott Hahn. Efficient and Scalable Multiprocessor Fair Scheduling Using Distributed Weighted Round-robin. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 65–74. ACM, 2009.

[47] Sam S Lightstone, Maheswaran Surendra, Yixin Diao, Sujay Parekh, Joseph L Hellerstein, Kevin Rose, Adam J Storm, and Christian Garcia-Arellano. Control Theory: a Foundational Technique for Self Managing Databases. In *Data Engineering Workshop, IEEE 23rd International Conference on*, pages 395–403. IEEE, 2007.

[48] Huan Liu. A Measurement Study of Server Utilization in Public Clouds. In *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, pages 435–442. IEEE, 2011.

[49] Google LMCTFY project (let me contain that for you), 2019. http://github.com /google/lmctfy (accessed September 1, 2019).

[50] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards Energy Proportionality for Large-scale Latency-critical Workloads. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 301–312, 2014.

[51] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 450–462. ACM, 2015.

[52] Memcached, 2019. https://memcached.org (accessed September 1, 2019).

[53] Mutilate: high-performance memcached load generator, 2019. https://github.com /andrewbartolo/mutilate (accessed September 1, 2019).

[54] Openstack Nova Compute Scheduler, 2019. https://docs.openstack.org/newton/config-reference/compute/schedulers.html (accessed September 1, 2019).

[55] OpenStack Software, 2019. https://www.openstack.org (accessed September 1, 2019).

[56] OProfile, 2019. https://oprofile.sourceforge.io (accessed September 1, 2019).

[57] Chandandeep Singh Pabla. Completely Fair Scheduler. *Linux Journal*, 2009(184), August 2009.

[58] Pradeep Padala, Kai-Yuan Hou, Kang G Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated Control of Multiple Virtualized Resources. In *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, pages 13–26. ACM, 2009.

[59] Pradeep Padala, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive Control of Virtualized Resources in Utility Computing Environments. In *Proceedings of the 2nd ACM European Conference on Computer Systems*, EuroSys '07, pages 289–302. ACM, 2007.

[60] Abhay K Parekh and Robert G Gallager. A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: the Single-node Case. *IEEE/ACM transactions on networking*, (3):344–357, 1993.

[61] Pearson Correlation Coefficient, 2019. https://en.wikipedia.org/wiki/Pearson_correlation_coefficient (accessed September 1, 2019).

[62] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-Aware Thread Management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 145–160, 2018.

[63] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 7:1–7:13, 2012.

[64] POSIX rlimits, 2019. http://man7.org/linux/man-pages/man2/getrlimit.2.html (accessed September 1, 2019).

[65] Jeff Roberson. ULE: A Modern Scheduler For FreeBSD. In *Proceedings of BSDCon'03*, pages 17–28. USENIX, 2003.

[66] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. CloudScale: Elastic Resource Scaling for Multi-tenant Cloud Systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 5:1–5:14. ACM, 2011.

[67] Paul Turner, Bharata B Rao, and Nikhil Rao. CPU Bandwidth control for CFS. In *Linux Symposium*, volume 10, pages 245–254, 2010.

[68] Vladimir Vapnik. *The Nature of Statistical Learning Theory.* Springer science & business media, 2013.

[69] Arunchandar Vasan, Anand Sivasubramaniam, Vikrant Shimpi, T Sivabalan, and Rajesh Subbiah. Worth Their Watts? - An Empirical Study of Datacenter Servers. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–10. IEEE, 2010.

[70] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache Hadoop Yarn: Yet Another Resource Negotiator. In *Proceedings of the 4th ACM Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, 2013.

[71] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient Performance Prediction for Large-scale Advanced Analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, 2016.

[72] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, page 18. ACM, 2015.

[73] Vincent M Weaver. Linux perf_event Features and Overhead. In *The 2nd International Workshop on Performance Analysis of Workload Optimized Systems, FastPath*, page 80, 2013.

[74] Chee Siang Wong, Ian Tan, Rosalind Deena Kumari, and Fun Wey. Towards Achieving Fairness in the Linux Scheduler. *ACM SIGOPS Operating Systems Review*, 42(5):34–43, 2008.

[75] Song Wu, Songqiao Tao, Xiao Ling, Hao Fan, Hai Jin, and Shadi Ibrahim. iShare: Balancing I/O Performance Isolation and Disk I/O Efficiency in Virtualized Environments. *Concurrency and Computation: Practice and Experience*, 28(2):386–399, 2016.

[76] Pengcheng Xiong, Yun Chi, Shenghuo Zhu, Hyun Jin Moon, Calton Pu, and Hakan Hacigümüş. Intelligent Management of Virtualized Resources for Database Systems

in Cloud Environment. In *2011 IEEE 27th International Conference on Data Engineering*, pages 87–98. IEEE, 2011.

[77] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple Linux Utility for Resource Management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[78] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI^2: CPU Performance Isolation for Shared Compute Clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*, pages 379–391. ACM, 2013.

[79] Yong Zhao, Kun Suo, Xiaofeng Wu, Jia Rao, Song Wu, and Hai Jin. Preemptive Multi-Queue Fair Queuing. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 147–158. ACM, 2019.

APPENDICES

Appendix A

Evaluation of the Fuzzy Control Algorithm

A.1 Experiment Results

This appendix presents the detailed performance of the fuzzy-control algorithm with three different workloads. The three workloads are the same as those in Section 4.4. This appendix complements the results presented in Section 4.4, where A.1.1 corresponds to 4.4.1, and A.1.2 to 4.4.2, and A.1.3 to 4.4.3 for full comparison purposes. The fuzzy-control results with a low-frequency and large-magnitude dynamic workload have been shown in Section 4.4.4.

A.1.1 Stable Workload

The results with the fuzzy-control algorithm are similar to the results with the hill-climbing algorithm in Section 4.4.1. The average throughput is 96.8% of the baseline result. The coefficient of variation of the throughput result is 0.03. The average numbers of cores after convergence with the two different initial allocations are 17.26 and 17.40 respectively. The coefficients of variation in the two cases are both less than 0.02.

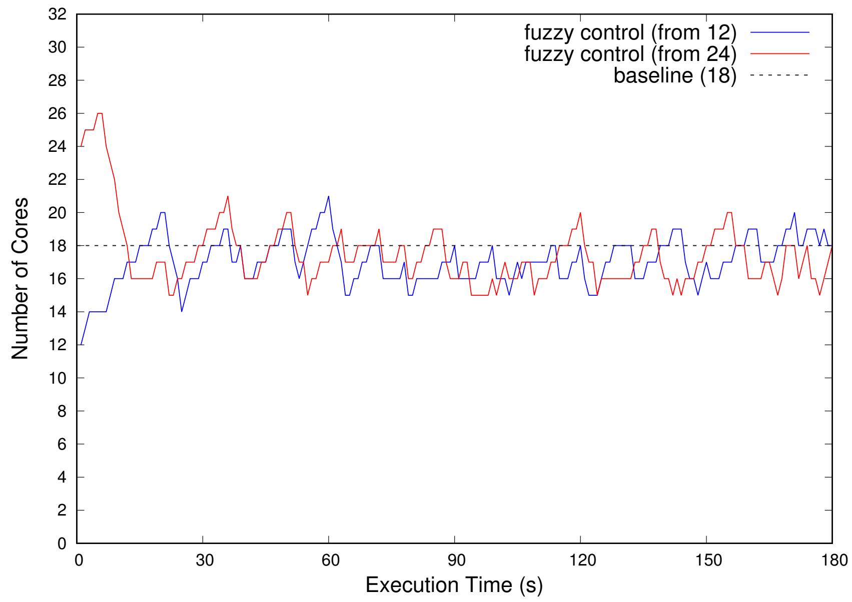
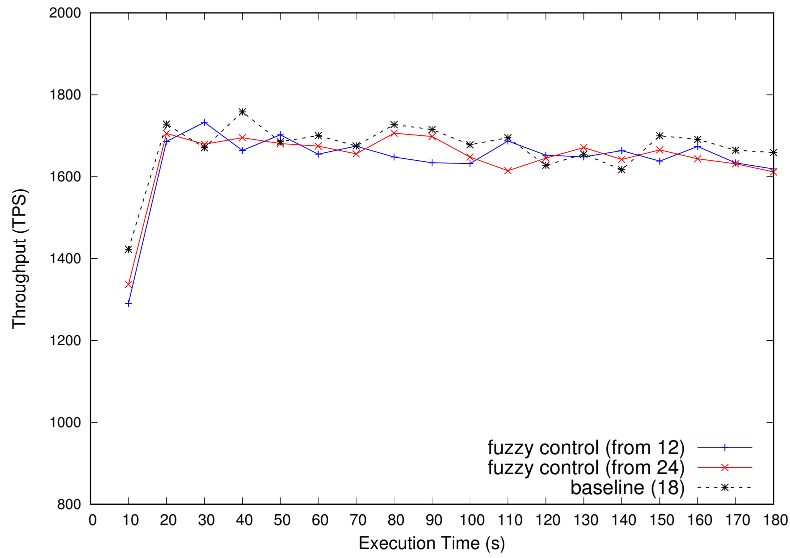
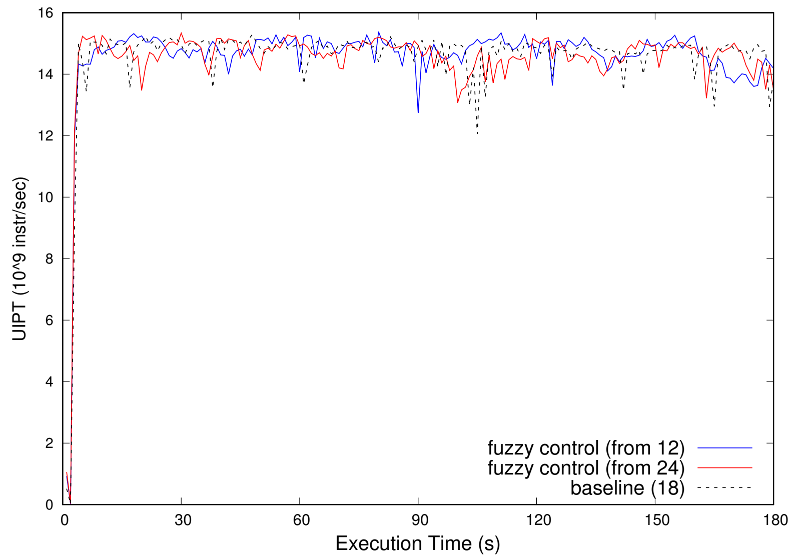


Figure A.1: Core Allocation with Fuzzy Control for Stable TPC-E Workload



(a) Throughput Result for TPC-E Workload

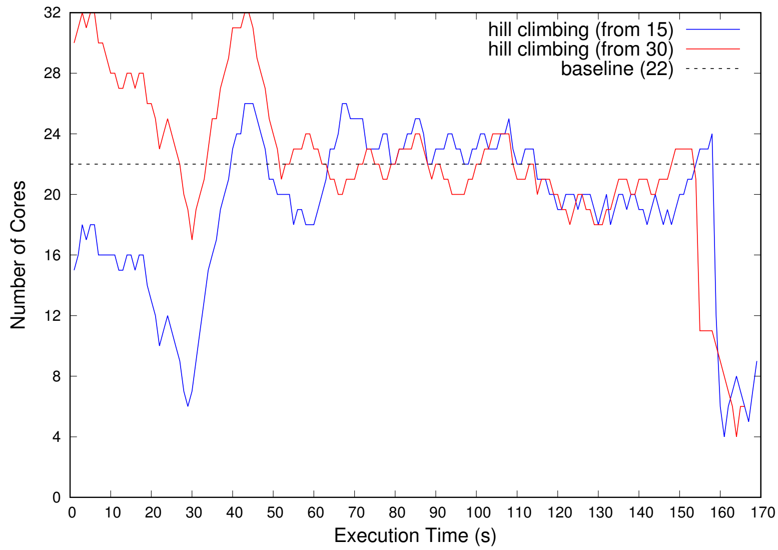


(b) UIPT Changes for TPC-E Workload

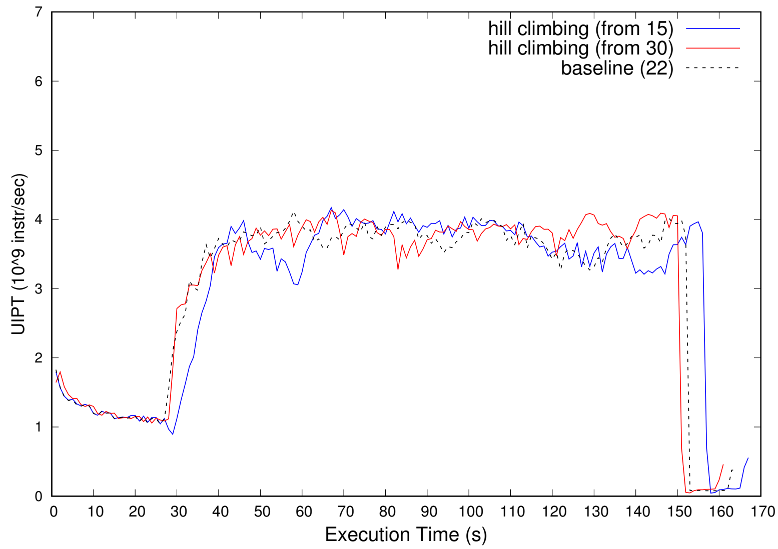
Figure A.2: Results with Fuzzy Control for Stable TPC-E Workload

A.1.2 Unsaturated Workload

With the Canneal workload, the results with the fuzzy-control algorithm are also similar to that with the hill-climbing algorithm in Section 4.4.2. When the initial allocation is 15 effective cores, the average execution time is 6% longer than baseline result, and the average quota value after convergence is 20.67. When the initial allocation is 30 effective cores, the average execution time is only 2% longer than the baseline result, and the average quota value after convergence is 20.68. In the two cases, the coefficients of variation of the execution time are less than 0.01. The coefficients of variation of the quota value are not greater than 0.04.



(a) Core Allocation for Canneal Benchmark



(b) UIPT Changes for Canneal Benchmark

Figure A.3: Results with Fuzzy Control for Canneal Benchmark

A.1.3 High-Frequency Dynamic Workload

The following figures show how the fuzzy-control algorithm performs with the workload in Section 4.4.3. Similar to the hill-climbing algorithm, the fuzzy-control algorithm achieves 95% of the baseline throughput with an initial allocation of 3 or 12 cores. The average numbers of cores after convergence are 7.16 and 7.25 respectively in the two cases.

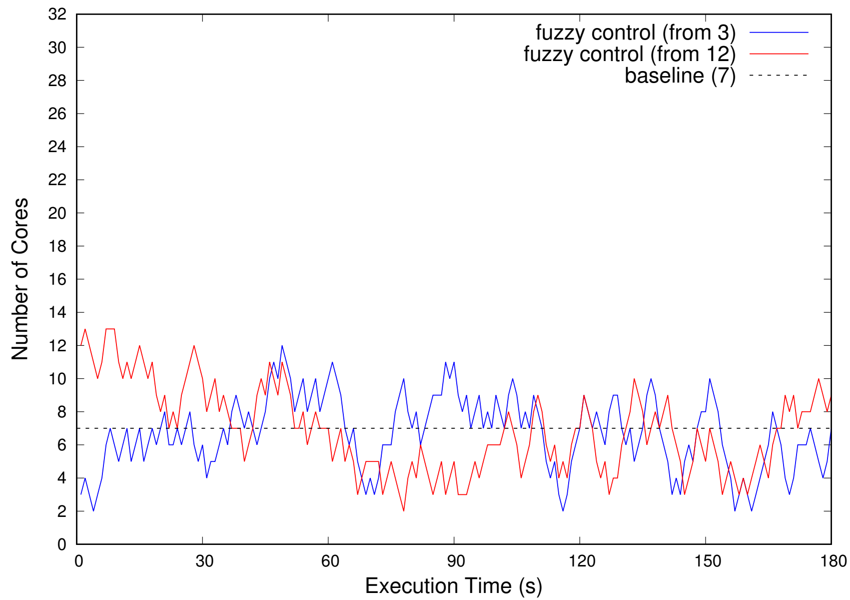
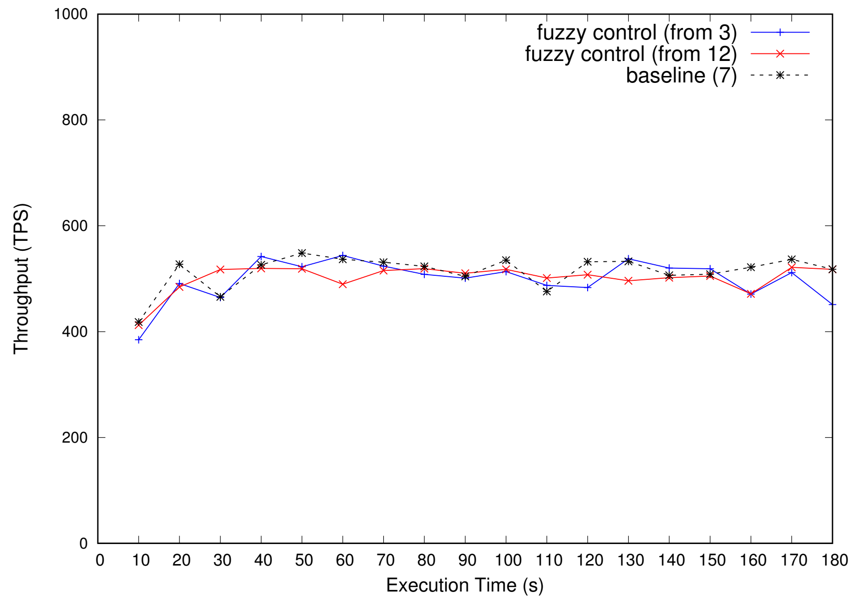
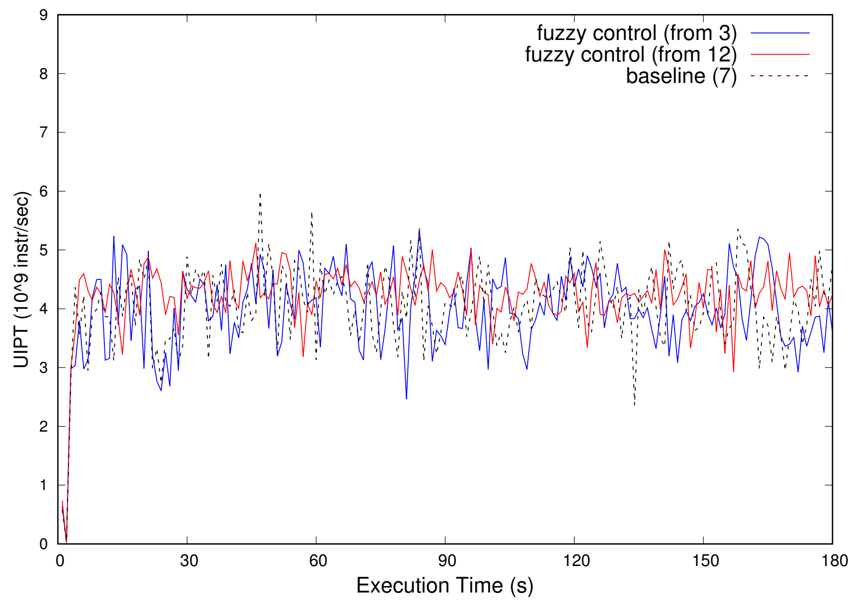


Figure A.4: Core Allocation with Fuzzy Control
for High-Frequency Dynamic TPC-E Workload



(a) Throughput Result for High-Frequency Dynamic TPC-E Workload



(b) UIPT Changes for High-Frequency Dynamic TPC-E Workload

Figure A.5: Results with Fuzzy Control for High-Frequency Dynamic TPC-E Workload

Appendix B

Evaluation of the Resource Manager

B.1 Proportional Sharing with Different Workloads

The experiments in Section 2.2.2 are repeated with the resource manager designed in this thesis. The proportional sharing policy is used in these experiments. As in Section 2.2.2, three experiments use two cpuhog workloads, two database workloads, and a combination of cpuhog and database workloads respectively. The three figures B.1, B.2 and B.3 correspond to those found in Section 2.2.2 respectively. The results show that the resource manager can effectively enforce resource isolation with different weight ratios while the weight mechanism in Linux cgroups cannot in the same experiments (cf. Section 2.2.2). The ideal results and the results with the quota mechanism are also shown in the following figures as a reference.

Two CPUhog Workloads

This experiment corresponds to that shown in Figure 2.8.

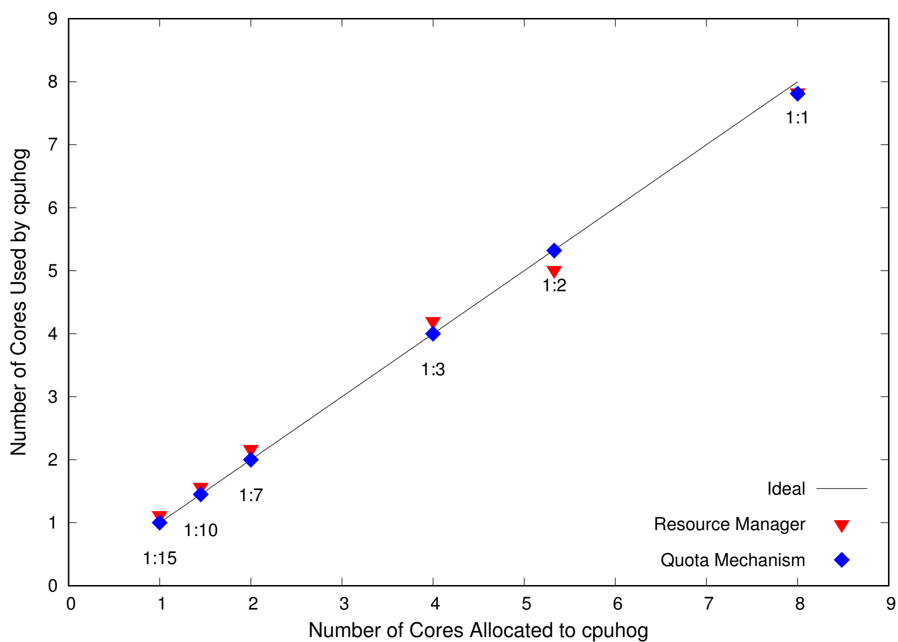


Figure B.1: CPU Usage of the CPUhog Workload with the Lower Weight

Two Database Workloads

The following figure corresponds to Figure 2.9.

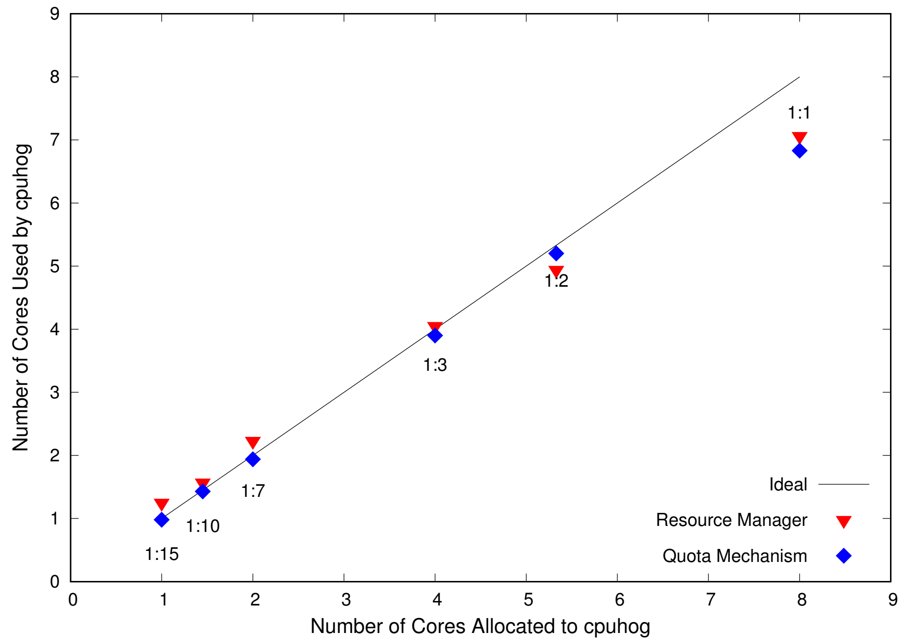
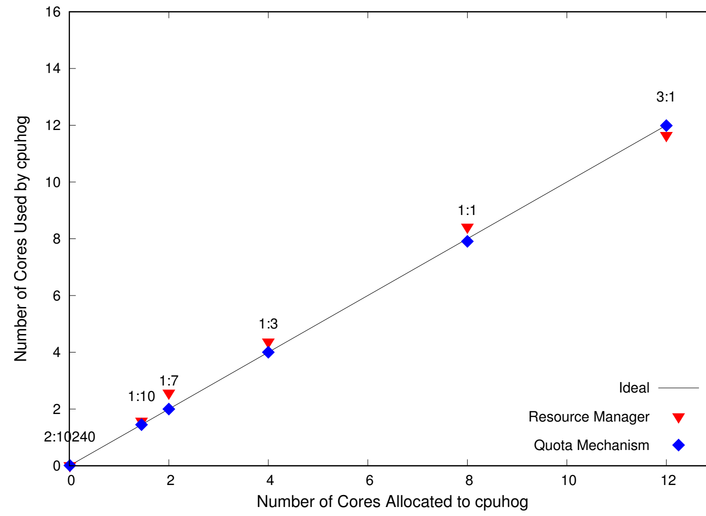


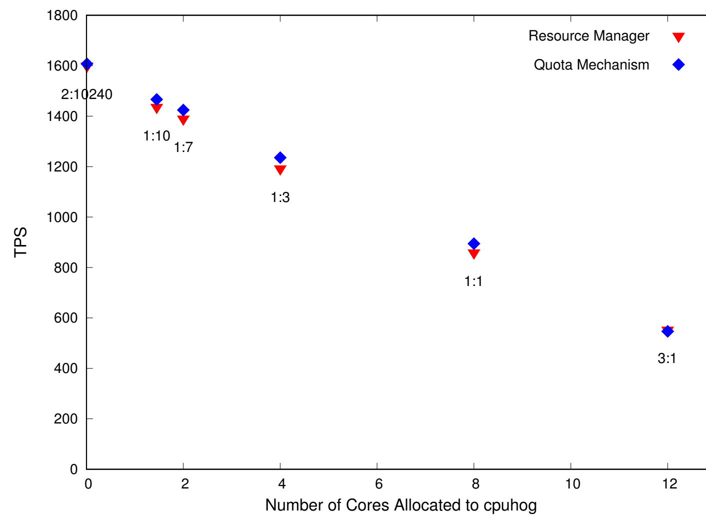
Figure B.2: CPU Usage of the TPC-E Workload with the Lower Weight

CPUhog and Database Workloads

The following results are associated with the results shown in Figure 2.10



(a) CPU Usage of the CPUhog Workload



(b) Throughput Results of the TPC-E Workload

Figure B.3: Results with Different Workloads